VERIFICATION OF CONCURRENT PROGRAMS

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

By

J. CHERIYAN

to the

COMPUTER SCIENCE PROGRAMME

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JUNE, 1983

6 JUN 1984

CENTRAL LIBRARY
I. I. T. Kanpur

Acc. No. A 82795

~~CSP M 190~~

CSP-1983-M-CHE-VER

CERTIFICATE

This is to certify that the thesis entitled, "VERIFICATION OF CONCURRENT PROGRAMS", has been carried out under my supervision by J.Cheriyan and that it has not been submitted elsewhere for a degree.

S. Biswas.

(Dr.S.Biswas)
Lecturer
Computer Science Programme
Indian Institute of Technology
Kanpur

Kanpur.

June, 1983.

ACKNOWLEDGEMENTS

I am grateful to Dr.S.Biswas for suggesting this topic initially and for guiding the work. I also benefitted from the discussions with him, during which many points were clarified.

I would also like to thank Mr.U.S.Mishra for his excellent typing.

-J. Cheriyan

CONTENTS

LIST OF SYMBOLS USED

$\wedge$	And
$\vee$	Or
$\neg$	Not
$\supset$	Implies
$\equiv$	Equivalent to
$\forall$	Universal Quantifier. For all.
$\exists$	Existential Quantifier. There exists.
P^x_e	The formula obtained by replacing every free occurrence of x, in formula P, by e.
$\{a,b,c\}$	Set of elements a,b,c.
$\in$	Set membership.
$\subset$	Subset of.
$\leadsto$	Leads to.
$\triangleq$	Defined as.
$\langle S \rangle$	The bracketed program statement or expression, S, is an indivisible action
$\vdash P$	P is a theorem in the [MP] system.
$\Box$	Henceforth - Temporal Operator.
∇	Eventually - Temporal Operator.

TEMPORAL LOGIC

Theorems.

$$\Box P \equiv \neg \nabla \neg P \qquad (\Box \text{ and } \nabla \text{ are duals})$$

$$\nabla P \vee \Box \neg P$$

$$\Box(P \wedge Q) \equiv \Box P \wedge \Box Q$$

$$\nabla(P \vee Q) \equiv \nabla P \vee \nabla Q$$

$$\Box(P \vee Q) \supset \Box P \vee \nabla Q$$

$$\Box P \wedge \nabla Q \supset \nabla[\Box P \wedge Q]$$

$$\nabla \Box P \wedge \nabla \Box Q \supset \nabla \Box(P \wedge Q)$$

$$[\Box P \wedge \Box(P \supset Q)] \supset \Box Q$$

$$[(P \wedge \Box Q) \supset \nabla R] \equiv [(P \wedge \Box Q) \supset \nabla(R \wedge Q)]$$

$$[(P \wedge \Box Q) \supset \nabla R] \supset [(P \wedge \Box Q) \supset \nabla(R \wedge \Box Q)]$$

$$[(P \supset \nabla R) \wedge (Q \supset \nabla R)] \supset [(P \vee Q) \supset \nabla R]$$

Rule of Generalization.

If temporal assertion P is a theorem, then $\Box P$ is also a theorem.

From this follows,

If $P \supset \nabla Q$ is a theorem, then $\Box(P \supset \nabla Q)$ is also a theorem ;

If $\Box P \supset \nabla Q$ is a theorem, then $\Box P \supset \Box \nabla Q$ is also a theorem.

(P, Q, R above are any temporal / immediate assertions).

ABSTRACT

Many of the well-known properties of concurrent programs can be classified as Safety Properties or Liveness Properties. In this thesis, several formal methods to derive Safety Properties and Liveness Properties of concurrent programs have been studied.

Various example programs have been treated using these methods, including an On-the-fly Garbage Collector and the Alternating-Bit Protocol.

Finally, a brief comparison is made, of the methods studied.

INTRODUCTION

When a program is viewed as an abstract object, it is of interest to know whether the program possesses certain desirable properties, which could be:

Partial Correctness-no execution of the program halts with a wrong result.

Termination-for a specified input data set, every execution of the program halts.

Mutual Exclusion-in a multiprocess program, two critical sections are not accessed together.

Deadlock Freeness-no execution of the program enters a set of states from which further progress is impossible.

First-come First-served - if process p requests service before process q, then process q cannot be served before process p.

Several formal methods have been suggested to derive such properties of programs. In this thesis, we have looked into the methods to derive safety and liveness properties of concurrent programs.

A concurrent program is a program which uses cobegin statements.

A cobegin statement, cobegin $S_1 \| S_2 \| \ldots \| S_n$ coend, signifies the nondeterministic interleaving of the atomic actions (indivisible actions) of the statements $S_1, S_2, \ldots, S_n$. The immediate constituents of a cobegin statement (i.e. $S_1, \ldots, S_n$) are also called 'processes'.

Is there any difference between a sequential program and a concurrent program? That is, is there any reason to consider programs which use cobegin statements (i.e. concurrent programs) separately from those that do not (i.e. sequential programs)?

There is indeed one difference.

The concept of atomic actions (indivisible actions) is of no significance for a sequential program, but is of importance for a concurrent program. In so far as a sequential program is characterised completely by its input-output behaviour, it is of no consequence whether the sequential program is a single atomic action, or is composed of many atomic actions. The grain of indivisible actions is of no importance.

For example the programs

Program A: $\langle x:=2 \rangle$

Program B: $\langle x:=1;\ x:=x+1 \rangle$

Program C: $\langle x:=1 \rangle;\ \langle x:=x+1 \rangle$

are all equivalent, as far as input-output behaviour is considered.

(Note - Angle brackets are used to mark off atomic actions).

Even if a concurrent program is characterised completely by its input-output behaviour, the grain of the indivisible actions of the processes, is of consequence.

For example, consider

Program A: Cobegin $\langle x:=2 \rangle \parallel \langle x:=2 \rangle$ coend

Program B: Cobegin $\langle x:=2 \rangle \parallel \langle x:=1;\ x:=x+1 \rangle$ coend

Program C: Cobegin $\langle x:=2 \rangle \parallel \langle x:=1 \rangle;\ \langle x:=x+1 \rangle$ coend.

Program A and B are equivalent.

However program C differs from A.

Program A terminates with $\{x=2\}$. Program C terminates with $\{X=2 \vee x=3\}$.

(The execution sequence $\langle x:=1\rangle$ $\langle x:=2\rangle$ $\langle x:=x+1\rangle$ for program C, gives $x=3$ on termination).

Safety properties of a program state that nothing bad ever happens. Some examples are - partial correctness, mutual exclusion and deadlock-freeness. The methods of Manna-Pnueli [MP], Owicki-Gries [OG] and Lamport [LAM3], for deriving safety properties, are treated in the chapter on safety properties.

Liveness properties state that something does happen. Some examples are termination, starvation-freeness - i.e. every request for a non-shareable resource is granted, and (in a protocol system) message reception. The chapter on Liveness properties surveys the methods of Owicki-Lamport [OL], Manna-Pnueli [MP] and Lamport [LAM1]..

The next chapter examines two well-known examples in the Manna-Pnueli formalism.

The on-the-fly Garbage Collector [DIJ2], is a two process program to collect garbage in a list processing system. The fine grain of interleaving makes the correctness proof quite involved.

The Alternating Bit protocol system [SUN], is used to transmit messages reliably through a medium that may lose messages. In this system, process communication is by message passing, unlike process communication by central shared memory in

the other examples. The correctness proof is obtained by modelling the distributed system as one with central shared memory- that is, the transmission medium is modelled as a queue.
We note in passing, that there are other methods, not considered in this thesis, to reason about properties of concurrent programs - e.g. [KEL], [LIP], [LAM4]. Also that, programs posses properties which do not fall into the safety and liveness categories - e.g.

- Equivalence - one program is equivalent to another.
- An assertion, P, might possibly become true for a program execution.

Most of the proof methods that we consider in this thesis use the temporal logic formalism. We end this chapter with a very brief note on temporal logic, and how program properties are expressed in this formalism.

A brief description of the Linear Time Temporal Logic, which is used in concurrent program verification, follows. Further discussion of Temporal Logic (in the context of concurrent programs) may be found in [LAM2], [PNU], [BMP], [GPSS], [GUP]. This description is from [LAM2].

The well-formed formulas of Temporal Logic are called <u>temporal assertions</u>. The set of temporal assertions is obtained in the obvious way from a set of atomic symbols - called atomic predicates - together with the usual logic operators $\wedge$, $\vee$, and $\neg$, and the unary temporal operators '$\Box$' (henceforth) and '∇' (eventually). Temporal assertions that do not contain either of the temporal operators, $\Box$ and ∇, are called (immediate) assertions or predicates.

A predicate P represents a declarative statement about the present state of the system, i.e. P is true now. The temporal assertion $\square$ P represents the statement that P is true now and will always be true in the future. The temporal assertion ∇ P represents the statement that P is true now or will become true some time in the future.

Models: The semantics of Temporal Logic is defined by describing how temporal assertions are to be interpreted as statements about an underlying model.

A model M is a pair (S, Σ) where S is a set of states and Σ is a set of sequences of states and Σ satisfies the Tail Closure property.

Let $\sigma = s_0, s_1, s_2, \ldots.$ be a sequence of states.

Then σ^+ is defined to be

$\sigma^+ \triangleq$ if length of σ is more than 1 then $s_1, s_2, s_3, \ldots$ else σ.

i.e. the sequence obtained by deleting the first element of σ.

Extending this, σ^i is defined to be

$$\sigma^i = \left[\sigma^{i-1}\right]^+ \quad \text{where } \sigma^0 = \sigma.$$

i.e. the sequence obtained by deleting the first i elements of σ.

The set of sequences, Σ, must satisfy the following condition,

Tail Closure: If $\sigma \in \Sigma$ then $\sigma^+ \in \Sigma$.

A state $s \in S$ is defined to be a truth valued function on the set of atomic predicates.

i.e. State s: $\{$atomic predicates$\} \rightarrow \{$True, False$\}$.

In the context of programs, the model M is related to the program as follows.

The set of states, S, is taken to be the set of all conceivable states of the program. Ordinarily, a program state is taken to be any combination of values of program variables and program control locations. Such a program state does not differ from the definition of a state given above. Eg. for a program, with a variable y, which has the value 1 in state s, s('$y > 0$') = true, s ('$y > 1$') = false, s('$y > 2$') = false .... and so on for all the atomic predicates.

The set Σ represents all "possible execution sequences" of the program, starting in any conceivable state. Thus a sequence $\sigma = s_0, s_1, s_2 \ldots$ in Σ represents an execution sequence that starts in state s_0, performs the first program step to reach state s_1, performs the next program step to reach state s_2....etc. All sequences in Σ are infinite - for a finite execution sequence this is ensured by infinitely repeating the last state. That is, if the execution sequence terminates in n steps, in state s_n, then the corresponding sequence $\sigma \in \Sigma$ has $s_m = s_n$, for all $m > n$.

Intuitively, in a sequence $[\sigma = s_0, s_1, s_2 \ldots]$, s_i represents the program state at the i^{th} instant.

The set of all "possible execution sequences" of a program is defined as the set of all sequences of conceivable program states, $s_0, s_1, s_2, \ldots$ such that

(i) s_{i+1} Next s_i, for all i.

Next is the 'next state' relation on pairs of states, where s_j Next s_i means that starting in state s_i and executing one program step can put the program into state s_j. For a nondeterministic program there may be several possible next states s_j.

(ii) Fairness

No execution sequence may have an action forever enabled without ever occuring.

An action is enabled if control resides at it and its enabling predicate is true .

The set of all possible execution sequences of a program does posses the Tail Closure property.

Tail Closure, for program execution sequences implies that the set of all possible computations from a given state is completely determined by the state itself and not by the history of the computation in reaching that state.

Let $\sigma \in \Sigma$ be any sequence $S_0, S_1, S_2, \ldots$

The linear Time interpretation of temporal assertion P in the model $M = (S, \Sigma)$ is the mapping,

$P: \Sigma \rightarrow \{True, False\}$ defined inductively as follows,

- if P is an atomic predicate, then

$$P(\sigma) = S_0(P).$$

- if P is an immediate assertion (predicate) then its interpretation is defined in the obvious way, in terms of the interpretations of its constituents

$$[Q \vee R](\sigma) = Q(\sigma) \vee R(\sigma)$$
$$[Q \wedge R](\sigma) = Q(\sigma) \wedge R(\sigma).$$
$$[\neg Q](\sigma) = \neg Q(\sigma)$$

- if P is any temporal assertion, the interpretation of $\Box$ P and ∇ P is defined as follows,

$$\Box P(\sigma) = \forall n \geqslant 0 : P(\sigma^n)$$
$$\nabla P(\sigma) = \exists n \geqslant 0 : P(\sigma^n).$$

A temporal assertion is M-valid for a model $M=(S,\Sigma)$ in the logic of linear time $[$i.e. $M \models P]$ if $P(\sigma)$ is true for every $\sigma \in \Sigma$.

In the linear time temporal logic, ∇ is equivalent to $\neg\Box\neg$, so that only the single temporal operator $\Box$ need be considered. The operator $\Box$ cannot express certain important properties of concurrent programs-such as First Come First Served.

"Generalized Temporal Logic" uses the dyadic $\Box$ operator instead of monadic $\Box$. The generalized temporal assertion R $\Box$ P represents the statement that P is true "as long as" the temporal assertion R remains true.

Formally, the meaning of R $\Box$ P is defined by extending the interpretation

$$R \Box P(\sigma)$$
$$\forall n \geqslant 0 : \left[\left(\forall i \in \{0,1,\ldots,n\} : R(\sigma^i) \right) \supset P(\sigma^n) \right].$$

Generalized temporal logic is as expressive as ordinary

temporal logic follows from,

$$\text{true} \ \Box \ P \equiv \Box P.$$

Some common program properties, expressed as temporal assertions follow - (from [GPSS]).

(i) Partial correctness, for a statement S with single-entry point l_o and single exit point l_e, (i.e. $\{P\}$ S $\{Q\}$),

$$\text{at } l_o \wedge P \supset \Box(\text{at } l_e \supset Q).$$

The post assertion Q may involve initial data values. Supposing $\bar{y}$ to be the vector of all data variables,

$$\text{at } l_o \wedge \bar{y} = \bar{y}_o \wedge P(\bar{y}_o) \supset \Box(\text{at } l_e \supset Q(\bar{y}_o, \bar{y})).$$

(ii) Total correctness for a single-entry, single-exit statement,

$$\text{at } l_o \wedge \bar{y}=\bar{y}_o \wedge P(\bar{y}_o) \supset \nabla(\text{at } l_e \wedge Q(\bar{y}_o, \bar{y}))$$

-being at l_o with P true and initial data values equal to $\bar{y}_o$ is guaranteed to lead to l_e with Q true.

(iii) A request (for some resource) is indicated by P being true. The response to this request is indicated by Q being true. The response may free the requester from being frozen in the requesting state - i.e. after the response, P need not be true.

Three kinds of response are possible

(a) Response to insistence

$$\Box P \supset \nabla Q.$$

This does not say that P must be true forever to get a response. It says that it is impossible for P to be forever true, without getting a response

i.e. it is equivalent to $\neg\Box(P \wedge \neg Q)$.

P may have to be true for an unbounded amount of time, to obtain a response.

(b) Response to persistence

$$\Box\nabla P \supset \nabla Q$$

- P need not be true continuously, but it may have to be true infinitely often.

(c) Response to impulse,

$$P \supset \nabla Q.$$

(d) Absence of unsolicited response, (Q is preceded by P)

$$\nabla Q \supset (\neg P \;\Box\; \neg Q) \wedge \nabla P$$

-If Q occurs at all, it is preceded by P. The consequent says that $\neg Q$ holds "as long as" P is not true and P does become true at some time. This is the general way of expressing "some P precedes some Q".

CHAPTER 1

METHODS FOR DERIVING SAFETY PROPERTIES

Safety properties state that nothing bad ever happens, or alternatively, whatever happens is good. Safety properties, expressed as temporal assertions, have three general forms -

(i) Init $\supset \Box$ I (invariant assertions)

Typical examples

Partial Correctness.

Mutual Exclusion.

(ii) $I \wedge \Box R \supset \Box I$

- Once I becomes true, it remains true forever, provided R is forever true.

  Such safety properties are very useful in deriving liveness properties. In this context, I is some desirable condition for progress and $\Box$ R states that no progress ever occurs. Thus from the above assertion, if the desirable condition ever becomes true and progress never occurs, then the desirable condition remains forever true. The obvious contradiction argument may then be used. The above assertion is equivalent to,

$$I \supset \Box I \vee \nabla \neg R$$

(iii) $I \supset R \Box I$

This is a stronger version of (ii). It uses the dyadic $\Box$ operator and says that if I ever becomes true, then it remains true "as long as" R remains true.

Example

First Come First Served: If process p requests service before process q, then process q cannot be served before process p.

Let pFIRST $\triangleq$ p is waiting for service and q is neither waiting for service nor being served.

pFIRST $\supset$ pWAITING $\square$ $\neg$(qSERVED).

Three methods of deriving safety properties due to, [MP]Z. Manna and A.Pnueli, [OG] S.Owicki and D.Gries, [LAM3] L.Lamport, are surveyed in this Chapter.

The first method uses an operational model to characterise programs. The last two methods are axiomatic and based on the Hoare Logic for sequential programs.

1.1 Manna-Pnueli Method

In this method every elementary statement is taken to represent a state to state transition. A concurrent program (in this method) has a fixed number of processes. Each process is a directed graph, whose nodes are called control locations and arcs are called transitions. Each transition is an indivisible action. Each transition α, has an enabling predicate c_α, which must be true for the transition to occur, and a location transformation function r_α, as well as a variable updating function f_α.

A concurrent program has the form

$$(\bar{y} := f_0\ (\bar{X}))\,[P_1 \parallel P_2 \parallel \ldots \parallel P_m]$$

The vector $\bar{X} = (X_1, X_2, \ldots, X_k)$ contains the input data. The vector $\bar{y} = (y_1, y_2, \ldots, y_n)$ is the vector of shared program variables. In fact, all program variables are shared, there are no variables local to a process. P_i, for $1 \leqslant i \leqslant m$, are the processes which constitute the program. There is also a vector $\bar{\pi} = (\pi_1, \pi_2, \ldots, \pi_m)$ of location variables. Each location variable, π_i, is the program counter for process P_i. At any instant, the value of each π_i is the name of some node in the directed graph which is process P_i. As control moves from node to node of process P_i, the location variable π_i is updated accordingly.

A transition α in a process P_j has the form,

$$\textcircled{l} \xrightarrow{\; c_\alpha(\bar{y}) \longrightarrow [\bar{y} := f_\alpha(\bar{y})] \;} \textcircled{l'}$$

l and l' are the names of the source and destination nodes of transition α. The transition is enabled only if $c_\alpha(\bar{y})$ is true and $\pi_j = l$. f_α is a function body which describes the change in $\bar{y}$ as a result of the transition α. That is, f_α is an n-tuple of expressions, and each expression may depend on $\bar{y}$.

For example (Assume n=5)

l: if $y_1 + y_2 * y_3 > 0$ then $y_3, y_5 := y_4/y_5, y_3+y_2$ fi

l':

represents the transition

$$\textcircled{1}\xrightarrow{y_1 + y_2 * y_3 > 0 \longrightarrow [\bar{y} := (y_1, y_2, y_4/y_5, y_4, y_3+y_2)]}\textcircled{1'}$$

The location transformation function r_α, for transition α in process P_j, updates the value of location variable π_j to 1', but has no effect on π_i, $i \neq j$.

Hence $r_\alpha\ (\pi_1, \pi_2, \ldots, \pi_{j-1}, \pi_j, \ldots, \pi_m) = (\pi_1, \pi_2, \ldots, \pi_{j-1}, 1', \ldots, \pi_m)$. In sum, the transition α has the form

$$\bigcirc\xrightarrow{\text{at } 1 \wedge c_\alpha(\bar{y}) \longrightarrow [(\bar{\pi};\ \bar{y}) := (r_\alpha(\bar{\pi}); f_\alpha(\bar{y}))]}\bigcirc$$

where at $1 \triangleq \exists j,\ 1 \leq j \leq m : \pi_j = 1$, i.e. 'at 1' is true if some process is presently at 1.

MP treats only programs with a fixed number of processes, so that, a program with nested cobegins cannot be examined by this method.

For example,

if $\langle y_1 \rangle \wedge \langle y_2 \rangle$ then S fi,

where $\langle y_1 \rangle \wedge \langle y_2 \rangle \triangleq$ Cobegin

Fetch (y_1) ‖ Fetch (y_2)

Coend

$\langle$Value of y_1 $\wedge$ Value of $y_2\rangle$

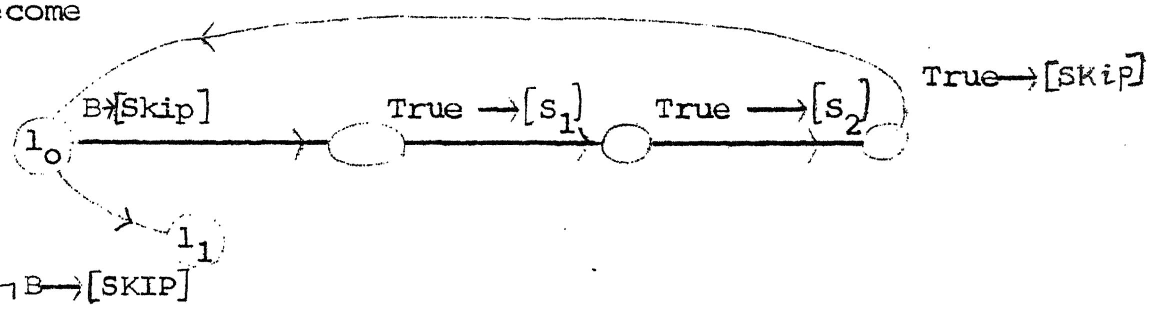
cannot be represented in MP .

However, any sequential construct with elementary statement or finer grain of interleaving, can be represented by transitions,

eg. l_o : While $\langle B \rangle$ do $\langle S_1 \rangle$; $\langle S_2 \rangle$ od

l_1 :

would become



Assertions are defined over program variables and location variables.

An assertion is an invariant for a program, according to [MP], if it is maintained by every transition and is initially true. For a program with input data satisfying the assertion $\emptyset\ (\bar{x})$, an assertion $Q\ (\bar{\pi};\ \bar{y})$ may be derived to be an invariant, by the following principle -

Invariance Principle.

Let $Q\ (\bar{\pi},\ \bar{y})$ be a state property of a program, (Note - Q has no temporal operators), such that

A: Q is initially true,

$$\text{I:}\quad \text{at } \bar{l}_o \wedge \emptyset\ (\bar{x}) \supset Q\ (\bar{l}_o;\ f_o\ (\bar{x}))$$

holds, where $\bar{l}_o = (l_o^1,\ l_o^2, \ldots, l_o^m)$ is the vector of initial locations.

B: Q is inductive for the program. That is, Q is preserved by every transition.

The Verification Condition

$$V_\alpha : (\text{at } l \wedge C_\alpha(\bar{y}) \wedge Q(\bar{\pi};\ \bar{y})) \supset Q\ (r_\alpha\ (\bar{\pi});\ f_\alpha\ (\bar{y}))$$

holds for every transition α in the program.

Then

$$\models \text{at } \bar{l}_o \wedge \emptyset\ (\bar{x}) \supset \Box Q\ (\bar{\pi};\ \bar{y})$$

may be deduced.

Example:

Semaphore Variable Rule.

A semaphore variable, say y, is initialised to a non-zero value, and can then be accessed only through Wait and Signal Operations.

Derive: $\models y \geqslant 0 \supset \square y \geqslant 0$

Initial: $y \geqslant 0 \supset y \geqslant 0$ (because y is initially non-negative)

Inductive: The assertion $Q \triangleq y \geqslant 0$ must be shown to be inductive.

Wait Operation:

α is $\bigcirc \xrightarrow{y > 0 \rightarrow [y:=y-1]} \bigcirc$ C_α is $y > 0$. f_α is $y-1$.

V_α: $y \geqslant 0 \wedge y > 0 \supset y-1 \geqslant 0$, which is true.

Signal Operation:

α is $\bigcirc \xrightarrow{\text{True} \longrightarrow [y:=y+1]} \bigcirc$ C_α is 'true', f_α is $y+1$.

V_α: $\text{True} \wedge y \geqslant 0 \supset y+1 \geqslant 0$, which is true.

Since y is not affected by any other transition, i.e. $f_\alpha(y) = y$ for all other transitions α, Q is inductive.

From this, $\models y \geqslant 0 \supset \square y \geqslant 0$.

The initial value of y must be non-negative, so that

$$\models \square y \geqslant 0.$$

1.1.1 Producer-Consumer Example

The producer-consumer example is a well known example in concurrent programming. It will be examined using the three methods of [MP], [OG], [LAM3] in this chapter. The [MP], [OG], proofs are from the original papers.

A producer computes values in sequence and passes them on to a consumer, which needs the values, in the same sequence, for its own computations. The two processes operate at roughly the same speeds, so it is profitable to interpose a buffer between them, to smooth out fluctuations of the individual process speeds. The buffer has a maximum capacity for N values. The producer repeatedly computes a value and puts it in the buffer, and the consumer repeatedly fetches a value from the buffer and does its own computation.

Program Producer-Consumer;

b: = NIL, S:=1, Ce:= N, cf: = 0

Producer;	Consumer;
l_o : Compute y_1	m_o: Wait (cf)
l_1 : Wait (ce)	m_1: Wait (s)
l_2 : Wait (s)	m_2: y_2:= Head (b)
l_3 : t_1:=b @ y_1	m_3: t_2:= Tail (b)
l_4 : b: = t_1	m_4: b: = t_2
l_5 : Signal (s)	m_5: Signal (s)
l_6 : Signal (cf)	m_6: Signal (ce)
l_7 : go to l_o	m_7: Compute using y_2
	m_8: go to m_o

Three semaphore variables S, cf, ce and three sequence variables b, t_1, t_2 are used.

s is a mutual exclusion semaphore to provide exclusive access to locations (l_3, l_4, l_5) and (m_2, m_3, m_4, m_5) for the producer and the consumer, respectively. The semaphore ce (count of empties) counts the number of free slots in buffer b. The semaphore cf (count of fulls) counts the number of items currently in the buffer b.

The permissible operations on a sequence variable b, are Head (b), which gives the first element of the sequence, Tail (b), which gives the rest of the sequence, and b @ y which extends the sequence by appending value y. The length of a sequence variable, b, is denoted by $|b|$. A sequence variable can be assigned the value of another sequence variable.

The initial condition, Init is

$$\text{Init} \triangleq \text{at } l_0 \wedge \text{at } m_0 \wedge (b=\text{NIL}) \wedge (S=1) \wedge (ce=N) \wedge (cf=0)$$

From the semaphore variable rule follows

$$\models \Box ((S \geq 0) \wedge (cf \geq 0) \wedge (ce \geq 0))$$

Exclusive access to the critical sections

$L = l_3, l_4, l_5$ and $M = m_2, m_3, m_4, m_5$ may be expressed as $\models \Box \neg (\text{at } L \wedge \text{at } M)$, or as

$\models \Box (\text{at } L + \text{at } M \leq 1)$ (here, truth values are numerically interpreted, with true=1, false=0).

This can be proved by showing the invariance of,

$$Q1: \text{at } L + \text{at } M + S = 1.$$

Initially at l_0=at m_0 = 1 which implies that at L = at M = 0. Also S=1. Hence, Q1 is initially true.

Next Q1 must be shown to be inductive, i.e. preserved by every transition of the program. This can be done by checking all transitions that modify the value of s or modify at L, at M. The only such transitions are those at l_2, l_5, m_1, m_5 (i.e., $l_2 \to l_3$, $l_5 \to l_6$, $m_1 \to m_2$, $m_5 \to m_6$). Consider the transition at l_2 - it decreases s by 1, but changes at L from 0 to 1, thus preserving Q1, similarly the transition at m_5 preserves Q1, because it increases s by 1 and also changes at M from 1 to 0. The other two transitions also preserve Q1, so that, by the Invariance Principle

$$\models \Box\ Q1 .$$

(implicitly assuming Init)

From $\Box Q1$ and $\Box\ s \geq 0$ (semaphore variable rule) follows that at L and at M can never be true together.

Proper buffer management can be shown by deriving

$$\models \Box \quad 0 \leq |b| \leq N.$$

Two invariant, assertions are required,

$$Q2:\ cf + ce + at\ l_{2..6} + at\ m_{1..6} = N$$

$$Q3:\ cf + at\ l_5, l_6 + at\ m_{1..4} = |b| .$$

(Note at $l_{2..6}$ or at $l_2..l_6$ stands for at $\{l_2,\ldots,l_6\}$ and similarly at $m_{1..4}$ or at $m_1..m_4$ stands for at $\{m_1,\ldots,m_4\}$). Q2 is true initially because cf = 0, ce = N and both at $l_{2..6}$ - and at $m_{1..6}$ are 0.

The only transitions affecting Q2 are those at l_1, l_6, m_0, m_6. Again - similarly to Q1 - these transitions preserve Q2. Hence,

$\models \Box$ Q2 (implicitly assuming Init)

Q3 is initially true because $|b| = 0$, cf $= 0$ and at l_5, l_6 and at $m_{1..4}$ are both 0. The transitions affecting Q3 are those at l_4, l_6, m_o, m_4. The transitions at l_6, m_o preserve Q3. However, the transitions at l_4, m_4 change the value of sequence variable b to t_1 and t_2 respectively. Hence, two additional invariant assertions,

$$\text{Q4:} \quad \text{at } l_4 \supset (|t_1| = |b| + 1)$$
$$\text{Q5:} \quad \text{at } m_4 \supset (|t_2| = |b| - 1)$$

are required.

Q4, Q5 are initially true because both antecedents are false.

Q4 may be falsified only by the transition at l_3, and this transition makes both at l_4 and $(|t_1| = |b| + 1)$ true.

Similarly, Q5 remains true after the transition at m_3, which is the only transition that can falsify Q5.

Hence

$\models \Box$ Q4 ,

$\models \Box$ Q5 by Invariance Principle.

Next, considering the transition at l_4 and Q3, it increases at l_5, l_6 by 1 and also increases $|b|$ by 1 (from $\Box$ Q4, $|t_1| = |b| + 1$ before this transition occurs) thus preserving Q3.

Similarly by $\Box$ Q5, the transition at m_4 preserves Q3.

Hence, $\models \Box$ Q3.

From Q3, $|b|$ is always the sum of non-negative values, so that $\models \Box\,(b \geqslant 0)$.

Further, it is always true that,

$$|b| - cf = at\ l_5, l_6 + at\ m_{1..4} \quad \text{(by Q3)}$$

$$\leqslant at\ l_{2..6} + at\ m_{1..6} \quad (\text{by } \{l_5, l_6\} \subset \{l_2, \ldots, l_6\},\ \{m_1, \ldots, m_4\} \subset \{m_1 \ldots, m_6\})$$

$$= N - cf + ce \quad \text{(by Q2)}$$

so that always $|b| - cf \leqslant N - cf + ce$

or

$$|b| \leqslant N - ce .$$

Hence, $\models \Box\, |b| \leqslant N$, because semaphore variable ce is always $\geqslant 0$.

This shows,

$$\models \Box\,(0 \leqslant |b| \leqslant N).$$

1.2 Owicki-Gries Method

The [OG] approach is to derive formulae $\{P\}\,S\,\{Q\}$, where S is a statement from an Algol-like language, extended to include the cobegin construct and a primitive construct, await. The await construct provides synchronisation and mutual exclusion. The notation $\{P\}\,S\,\{Q\}$ has exactly the same meaning as in Hoare Logic, i.e. if P is true before execution of S, Q is true after execution of S.

The axioms and inference rules for sequential constructs are the same as in Hoare Logic

NULL $\{P\}$ Skip $\{P\}$

ASSIGNMENT $\{P^X_E\}$ X: = E $\{P\}$

ALTERNATION $$\frac{\{P \wedge B\} S_1 \{Q\},\ \{P \wedge \neg B\} S_2 \{Q\}}{\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2\ \{Q\}}$$

ITERATION $$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\} \text{ While } B \text{ do } S\ \{P \wedge \neg B\}}$$

COMPOSITION $$\frac{\{P_1\}\ S_1\ \{P_2\}, \{P_2\}\ S_2\ \{P_3\}, \ldots\ldots, \{P_n\}\ S_n\ \{P_{n+1}\}}{\{P_1\} \text{ begin } S_1; S_2; \ldots S_n \text{ end } \{P_{n+1}\}}$$

CONSEQUENCE $$\frac{\{P_1\}\ S\ \{Q1\},\ P \vdash P_1,\ Q1 \vdash Q}{\{P\}\ S\ \{Q\}}$$

The notation $P \vdash P_1$ means it is possible to prove P_1 using P as an assumption, in a deductive system which is valid for the data types and operations used in the programming language.

A proof-outline is a program annotated with assertions, such that if a statement, S, occurs between two assertions P and Q, then $\{P\}$ S $\{Q\}$ is derivable. In a proof-outline, two adjacent assertions $\{P_1\}\{P_2\}$ denote a use of the rule of consequence, where $P_1 \vdash P_2$.

Each statement S is always preceded directly by an assertion called its precondition, written pre (s). Similarly the postcondition post (s), is the assertion following statement s.

For example, the program

S $\equiv$ begin X:=a; if e then S_1 else S_2 end, may have the proof outline,

$\{P\}$

begin $\{P\}$

$\{P_1{}^x_a\}$

X: = a;

$\{P_1\}$

if e then $\{P_1 \wedge e\}$

S_1

$\{Q_1\}$

else $\{P_1 \wedge \neg e\}$

S_2

$\{Q_1\}$

$\{Q_1\}$

$\{Q\}$

end

$\{Q\}$

Note - $P_1{}^x_a$ denotes P_1 with all free occurrences of x substituted by a.

In this proof-outline pre(X:=a) = $P_1{}^x_a$, pre $(S_2) = P_1 \wedge \neg e$, post (if e then S_1 else S_2) = Q_1, etc.

The cobegin statement has the form

Cobegin $S_1 \| S_2 \| \ldots \| S_n$ coend.

$S_1, S_2, \ldots S_n$ are statements of the programming language - each S_i may also be called a process.

Obviously, the indivisible actions of the processes S_i are of interest.

Each assignment statement and each expression is an indivisible action.

Thus the grain of interleaving is fixed at the elementary-statement/expression level. A finer grain of

interleaving of the memory reference may be assumed, if programs adhere to the following convention

- Each expression and assignment statement refers at most once to a single variable that is changed by another process.

For example, let a,b,c be variables that are changed only by process S_1, and X,y,z be variables that may be changed by several processes S_1, S_2, S_3,...

Then statements

$$\langle X \rangle := \langle a \rangle + \langle b \rangle - \langle c \rangle * 5 \text{ , } \langle b \rangle := \langle a \rangle - \langle c \rangle * \langle y \rangle,$$

within process S_1, satisfy the above convention.

The expression $\langle x \rangle + \langle y \rangle * \langle c \rangle$ does not satisfy the convention, as it refers to two changing variables x, y. Similarly, the assignment statement $\langle X \rangle := \langle x \rangle + \langle a \rangle * \langle b \rangle$ does not satisfy the convention, as it refers twice to changing variable x. The concurrent assignment statement $\langle x \rangle, \langle y \rangle :=$ $\langle a+b,\ b-c \rangle$ also does not satisfy the convention.

The following example shows why the convention is needed.

S_A : $\{X = 0\}$
Cobegin $\langle X := X + 2 \rangle \parallel \langle X := 3 \rangle$ coend
$\{X = 3 \vee X = 5\}$

Either $\langle X:=3 \rangle$ occurs last, in which case $\{X=3\}$ upon termination, or $\langle X:=3 \rangle$ occurs first, in which case $\{X = 5\}$ upon termination

S_B: $\{X=0\}$
Cobegin $\langle X \rangle := \langle X \rangle + 2 \parallel \langle X:=3 \rangle$ coend
$\{X = 2 \vee x=3 \vee x = 5\}$

In this example, $\langle X:=3 \rangle$ may occur between the two memory references of $\langle X \rangle := \langle X \rangle + 2$, so that $\{X=2\}$ upon termination.

Clearly $\langle X \rangle := \langle X \rangle + 2$ cannot be assumed to be an indivisible action, if the variable x is changed by other processes.

On the other hand, if an action has no more than one reference to a single changing variable, the whole action may be taken to occur indivisibly at the moment of that memory reference. This is because, by the convention, all other references by this action are to variables not changed by other processes, hence all such references can be 'translated in time' to the moment at which the changing variable is referred.

In sum, because of adherence to this convention, a memory reference level grain of interleaving, is equivalent to an elementary-statement/expression level grain of interleaving.

The cobegin statement is defined formally, by the rule

$$\text{COBEGIN} \quad \frac{\{P_1\}\ S_1\ \{Q_1\}, \ldots, \{P_n\} S_n \{Q_n\} \text{ are interference-free}}{\{P_1 \wedge \ldots \wedge P_n\} \text{ Cobegin } S_1 \,/\!/\, \ldots \,/\!/\, S_n \text{ Coend } \{Q_1 \wedge \ldots \wedge Q_n\}}$$

The interference-freeness of a set of formulae $\{P_j\}\ S_j\ \{Q_j\}_{1 \leq j \leq n}$, guarantees that the formula $\{P_i\}\ S_i\ \{Q_i\}$ derived for some S_i in isolation, remains valid, despite the interleaving of indivisible actions from S_j, $1 \leq j \leq n$, $j \neq i$.

Definition of non-interference. Given a proof-outline $\{P\}\ S\ \{Q\}$ and a statement T with precondition pre (T), T does not interfere with $\{P\}\ S\ \{Q\}$ if the following hold

(i) $\{Q \wedge pre(T)\}\ T\ \{Q\}$,

and (ii) Let S' be any statement of S but not within an <u>await</u>. Then,

$$\{Pre(S') \wedge Pre(T)\}\ T\ \{Pre(S')\}$$

Definition of interference-free. $\{P_1\}\ S_1\ \{Q_1\}, \ldots, \{P_n\}\ S_n\ \{Q_n\}$ are interference-free if the following holds. Let T be an <u>await</u> or assignment statement (which does not occur within an <u>await</u>) of process S_i. Then for all j, $j \neq i$, T does not interfere with $\{P_j\}\ S_j\ \{Q_j\}$.

An <u>await</u> statement has the form

Await B then S,

where B is a boolean expression.

The whole <u>await</u> statement is an indivisible action. An <u>await</u> statement cannot contain a cobegin or another <u>await</u>. The process within which the <u>await</u> occurs waits for the condition B to become true, and then performs S. No action from another process may be interleaved between the evaluation of B (to true) and the subsequent execution of S.

The formal definition of <u>await</u> is

$$\text{AWAIT} \quad \frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \text{Await B then S}\ \{Q\}}$$

Obviously, the statement S within an <u>await</u> need not adhere to the convention given above for reference to changing variables.

[OG] also uses Auxiliary variables. Auxiliary variables are used only for proof purposes, but not for the program itself.

No Auxiliary variable may occur on the right side of an assignment. Such variables serve two purposes.

(i) As location variables, to indicate where control is within a particular process.

(ii) As history variables to record the effect of the past computation on some variables; eg. the number of Wait or Signal operations on a semaphore variable.

The [OG] system without Auxiliary Variables is incomplete (ref.[OG2]). Auxiliary variable Transformation. Let AV be an auxiliary variable set (i.e. the set of Auxiliary variables) for S', and P and Q assertions that do not contain free variables from AV. Let S be obtained from S' by deleting all assignment statements with assignments to the variables in AV. Then

$$\frac{\{P\}\ S'\ \{Q\}}{\{P\}\ S\ \{Q\}}\ .$$

1.2.1 Example Producer-Consumer

The proof of a producer-consumer program, from [OG], is shown.

The program is to copy an array of values A[1..M] into an array B[1..M]. The producer must pass the values from array A, to the consumer, which puts them into array B. A buffer, of maximum capacity N, is interposed between producer and consumer. The buffer description is,

Buffer [0..N-1] is a shared array;

i = number of elements added to buffer;

j = number of elements removed from buffer;

The Buffer contains i-j values, in the order

Buffer [j mod N],..., Buffer [(i-1) mod N].

Two semaphores Full, Empty are used to synchronize access to the buffer. Empty gives the number of vaccant slots in the buffer, Full gives the number of occupied slots.

The semaphores are translated into awaits, using

Wait (sem) $\triangleq$ Await sem > 0 then sem: = sem-1,

Signal (sem) $\triangleq$ Await True then sem:= sem+1.

Auxiliary variables, which count the number of semaphore operations performed, are also introduced.

```
S:  Begin
        Full: = 0; Empty: = N; i:=1; j := 1;
        Cobegin
        Producer: While i ≤ M do
                  begin X: = A [i] ;
                        Wait (Empty);
                        Buffer [i mod N]: = X;
                        Signal (Full);
                        i: = i + 1
                  end
```

```
Consumer: While j ≤ M  do
          begin Wait (Full);
                y: = Buffer [j mod N];
                Signal (Empty);
                B [j]: =  y;
                j:=j + 1
          end
       Coend
    End
```

S': The Program with Auxiliary Variables and Awaits is,

```
Begin
    Full: =0, Empty: = N; i:=1; j:=1;
    Wfull, Sfull, Wempty, Sempty:= 0,0,0,0;
Cobegin
    Producer: While i ≤ M  do
              begin X: = A [i];
              Await Empty > 0  then
                    begin Empty: = Empty - 1;
                          Wempty: = Wempty + 1 end;
              Buffer [i mod N]: = X;
              Await True then
                    begin Full: = Full + 1;
                        Sfull : = Sfull + 1 end;
              i:=i + 1
              end
```

```
Consumer:  While j ≤ M do
           begin Await Full > 0  then
           begin Full: = Full - 1;
                 Wfull: = Wfull + 1 end;
           y:= Buffer [j  mod N] ;
           Await true then
           begin Empty: = Empty + 1;
                Sempty: = Sempty + 1 end;
           B[j]: = y;
           j: = j+1
        end
  Coend
End
```

Let I be the assertion,

$$I \triangleq (\text{Buffer}[k \bmod N] = A[k], \text{ for } K: \text{Sempty} < k \leq \text{Sfull})$$
$$\wedge \text{ Full} = \text{Sfull} - \text{Wfull}$$
$$\wedge \text{ Empty} = N + \text{Sempty} - \text{Wempty}$$
$$\wedge\ 1 \leq i \leq M+1$$
$$\wedge\ 1 \leq j \leq M+1.$$

I is the fundamental program invariant - it is not interfered with by any producer or consumer action.

The proof outline for the main program, using I, is

```
Begin
    Full: = 0; Empty: = N, i:=1; j:=1;
    Sfull, Wfull, Sempty, Wempty: = 0,0,0,0;
```

$$\{ I \wedge \text{Sfull} = \text{Wempty} \wedge i = \text{Sfull} + 1 \wedge \text{Sempty} = \text{Wfull}$$
$$\wedge\ j = \text{Sempty} + 1 \}$$

Cobegin

$\{I \wedge Sfull = Wempty \wedge i = Sfull + 1\}$

producer

$\{I\}$

$\|$

$\{I \wedge Sempty = Wfull \wedge j = Sempty + 1\}$

Consumer

$\{I \wedge (B[k] = A[k],\ 1 \leqslant k \leqslant M)\}$

Coend

end

$\{B[k] = A[k],\ 1 \leqslant k \leqslant m\}$.

The last assertion is indeed the desired output assertion, and says that array A has been fully copied into array B.

The auxiliary variables can now be removed, using the given inference rule, to yield a proof of

$$\{M \geqslant 0\}\ S\ \{B[k] = A[k],\ 1 \leqslant k \leqslant M\}.$$

The consumer proof outline follows:

IC is the assertion

$$IC \triangleq (B[k] = A[k],\ 1 \leqslant k < j).$$

$\{I \wedge IC \wedge Sempty = Wfull \wedge j = Sempty + 1\}$

Consumer: While $j \leqslant M$ do

begin

$\{I \wedge IC \wedge Sempty = Wfull \wedge j = Sempty + 1 \wedge j \leqslant M\}$

Await Full > 0 then

begin full: = full-1; Wfull; = Wfull + 1 end;

$\{I \wedge IC \wedge Sempty = Wfull-1 \wedge j = Sempty + 1 \wedge j \leqslant M\}$

y: = Buffer [j mod N];

$\{ I \wedge IC \wedge Sempty = Wfull - 1 \wedge j = Sempty + 1 \wedge j \leq M$
$\wedge y = A[j] \}$

Await true then

begin Empty: = Empty +1; Sempty:= Sempty +1 end;

$\{ I \wedge IC \wedge Sempty = Wfull \wedge j = Sempty \wedge j \leq M$
$\wedge y = A[j] \}$

B [j]: = y;

$\{ I \wedge IC \wedge Sempty = Wfull \wedge j = Sempty \wedge j \leq M$
$\wedge B[j] = A[j] \}$

j:=j+1;

$\{ I \wedge IC \wedge Sempty = Wfull \wedge j = Sempty + 1 \wedge j \leq M + 1 \}$

end

$\{ I \wedge IC \wedge j = M+1 \}$

$\{ I \wedge (B[k] = A[k], 1 \leq k \leq M) \}$

The producer proof outline is

$\{ I \wedge Sfull = Wempty \wedge i = Sfull + 1 \}$

Producer: While i $\leq$ M do

begin

$\{ I \wedge Sfull = Wempty \wedge i = Sfull + 1 \wedge i \leq M \}$

X:= A [i];

$\{ I \wedge Sfull = Wempty \wedge i = Sfull + 1 \wedge i \leq M \wedge X = A[i] \}$

Await Empty > 0 then

begin Empty: = Empty -1; Wempty:=Wempty+1 end;

$\{I \wedge Sfull = Wempty - 1 \wedge i = Sfull + 1 \wedge i \leqslant M \wedge x = A[i]\}$

Buffer [i mod N]: = x ;

$\{I \wedge Sfull = Wempty - 1 \wedge i = Sfull + 1 \wedge i \leqslant M \wedge Buffer[i \bmod N] = A[i]\}$

Await True then

begin Full: = Full+1; Sfull:=Sfull +1 end;

$\{I \wedge Sfull = Wempty \wedge i = Sfull \wedge i \leqslant M\}$

i: = i+1

$\{I \wedge Sfull = Wempty \wedge i = Sfull + 1 \wedge i \leqslant M+1\}$

end

$\{I \wedge i = Sfull + 1 = M + 1\}$

Interference-freedom is quite obvious.

Examining all consumer assertions, except for I, these assertions involve only variables used by the consumer.

Similarly, for all producer assertions, the consumer changes no variables, except those mentioned by I.

The assertion I is invariantly true in both processes.

1.3 Lamport's method

[LAM3] uses the following method to prove an invariant Q for a program S.

Assume $\vdash$ after (s) $\supset$ Q.

Now find a predicate P such that

(a) The Initial condition implies that P is true

(b) $\vdash \{P\}$ S $\{true\}$

(c) $\vdash P \supset Q$.

Of course, to derive $\{P\}$ S $\{true\}$ the entire program S is examined from the atomic actions upwards and the axioms and rules of inference are used.

A formula $\{P\}$ S $\{Q\}$ in [LAM3] has a different meaning from one in [OG]. P may depend on program control locations, in addition to the data variables. The elementary predicates for control locations of a particular statement S, are at('S'), in ('S') and after ('S'). at ('S') is true when control is at the beginning of (i.e. just before) S, in ('S') is true when either at ('S') is true or control is somewhere within the statement S, and after ('S') is true when control is at the location immediately following the statement S. A formula $\{P\}$ S $\{Q\}$ means that if execution is started anywhere in S with P true, then P is preserved, as long as control is in S, and Q becomes true upon termination of S. The constraint on P being preserved holds only for the actions of S - i.e. it does not prohibit some other process from falsifying P while S is executing.

Suppose S $\triangleq$ cobegin $S_1 \| S_2$ coend

Then $\vdash \{in\ (S_2)\}\ S_1 \{true\}$, because no action of S_1 can affect the control location of S_2 - but, while S_1 is still executing, S_2 may terminate, thus falsifying in (S_2).

In other words, some process running in parallel with S_1 may 'interfere' with the precondition of S_1. The cobegin statement inference rule gets around this difficulty.

Let B $\triangleq$ cobegin $S_1 \| S_2 \| \ldots \| S_n$ coend

Cobegin rule $\dfrac{\{P\}S_1\ \{P\},\{P\}S_2\ \{P\},\ \ldots,\{P\}\ S_n\ \{P\}}{\{P\}\ B\ \{P\}}$

i.e. in order to obtain a formula for a cobegin statement it must be shown that the assertion P is maintained by every substatement of the cobegin and P remains true even after any or all of the substatements have terminated. Thus the cobegin rule guarantees 'interference freedom' since P is preserved by every atomic action.

Unlike in [OG], where the elementary or atomic action is the memory reference, [LAM3] does not fix on any atomic action. If an action is atomic, its proof rule is exactly the same as that in Hoare Logic of sequential programs. Composite actions are decomposed into atomic actions - and the formula for the whole composite action is obtained by given rules of inference.

The treatment of expressions in [LAM3] is quite thorough. Typically an expression involves several variables (and constants), as well as several operations on them. Each occurrence of a variable is actually a reference to its memory location. This memory reference is explicitly modelled in [LAM3] by associating a 'value' attribute with each atomic sub-expression. In implementation terms, the 'value' of an atomic sub-expression is some private location (such as a register) into which the result of evaluating the expression is placed. The 'value' attribute of a sub-expression is not affected by any other action, as it is totally private.

So that (a) $\langle x := X+1\rangle$ (b) $\langle x\rangle := \langle X+1\rangle$ and (c) $\langle x\rangle := \langle X\rangle + 1$ are all different from each other. (a) is atomic. (b) is

equivalent to

$$\langle \text{value } (\text{'x+1'}) := X+1 \rangle;$$
$$\langle x := \text{value } (\text{'X+1'}) \rangle,$$

and c is

$$\langle \text{value } (\text{'x'}) := x \rangle;$$
$$\langle \text{value } (\text{'}\langle x \rangle + 1\text{'}) := \text{value } (\text{'x'}) + 1 \rangle;$$
$$\langle x := \text{value } (\text{'}\langle x \rangle + 1\text{'}) \rangle$$

[LAM 3] defines an arbitrary expression by the following rules

(a) $\langle e \rangle \longrightarrow \langle \text{value } (\text{'e'}) := e \rangle$

(b) $f\,(e_1, e_2, \ldots, e_n) \longrightarrow$

$$\text{cobegin } e_1 \,\|\, e_2 \,\|\, \ldots\ \|\, e_n \text{ coend;}$$
$$\langle \text{value } (\text{'f}(e_1, e_2, \ldots, e_n)\text{'}) :=$$
$$f(\text{value } (\text{'}e_1\text{'}), \text{value}(\text{'}e_2\text{'}), \ldots, \text{value}(\text{'}e_n\text{'})) \rangle.$$

Although the rules are simple enough, a system allowing this degree of interleaving does not always match the common sense interpretation.

e.g. While $\langle A \rangle \vee \neg \langle A \rangle$ do

S

od

The boolean expression $\langle A \rangle \vee \neg \langle A \rangle$ may become false if A changes from false to true between the two memory reference to A (assuming the first $\langle A \rangle$ is fetched first).

As a last example, the producer-consumer program is proved using [LAM3].

1.3.1 Example Producer-Consumer

```
Var  A,B : array [1..M] of T;
     Buffer: array[0..N-1] of T;
     i,j: integer; X,y; T; empty, full:semaphore;
     i: = 1; j:=1 ; empty:= N; full:=0;
cobegin
Producer: While l'o: ⟨i ≤ M⟩ do
                l0: ⟨x := A[i]⟩;
                l1 : ⟨wait (empty)⟩;
                l2 :⟨Buffer [i mod N]: = X⟩;
                l3 : ⟨Signal (full)⟩;
                l4 : ⟨i := i+1⟩
                         od
         ||
Consumer: While m'o: ⟨j ≤ N⟩ do
                m0: ⟨wait (full)⟩;
                m1: ⟨y: = Buffer [j mod N]⟩;
                m2: ⟨ signal (empty)⟩;
                m3: ⟨B[j]: = y⟩;
                m4: ⟨j: = j+1⟩
                        od
     coend
  End.
```

It must be shown that finally $A[k] = B[k], \forall k: 1 \leqslant k \leqslant M$. To do this it must be ensured that the buffer length is always between 0 and N. If the producer is about to access the buffer (at (l_2)) then the buffer length is between 0 and N-1.

whereas if the consumer is about to access the buffer (at (m_1)) then the buffer length is between 1 and N.

Usually the buffer length = i-j.

Also usually,

$$i-j = N-\text{empty}$$

(because whenever the producer increments i and overtakes the consumer, it also decrements 'empty')

and

$$i-j = \text{full}$$

(again because the producer increments i and signals 'full' when it overtakes the consumer).

full and empty, being semphores, are always $\geqslant 0$.

Hence,

$$i-j \leqslant N$$

and

$$i-j \geqslant 0.$$

Actually, there are several special cases which are examined in the proof.

Firstly, the atomic actions are examined

$$\{\text{true}\}$$

$$i: = 1;\ j: = 1:\ \text{empty}: = N;\ \text{full}: = 0;$$

$$\{i = j = 1 \wedge \text{empty} = N \wedge \text{full} = 0\}$$

Notation used

$$I1 \triangleq A[k] = B[k], \forall k: 1 \leqslant k < j$$

$$[j:i]\ \text{Buffer} = A \triangleq \text{Buffer}[k \bmod N] = A[k], \forall k: j \leqslant k \leqslant i$$

$$(j:i]\ \text{Buffer} = A \triangleq \text{Buffer}[k \bmod N] = A[k], \forall k: j < k \leqslant i$$

$$[j:i)\ \text{Buffer} = A \triangleq \text{Buffer}[k \bmod N] = A[k], \forall k: j \leqslant k < i$$

etc.

'Predicate' $\longrightarrow P_1, P_2 \triangleq$ if 'Predicate' is true, then P_1 holds, else P_2 holds.

The 'predicate' used will always be the control location predicate <u>at</u> .

Producer: $\{ i \leq M \;\wedge$

$\quad$ at $m_o \rightarrow i-j = full,\ i-j = full + 1$

$\wedge$ at $m_{o..2} \rightarrow i-j = N\text{-}empty,\ i-j = N+1 - empty$

$\wedge$ at $m_{o..2} \rightarrow [j:i)$ Buffer=A, $(j:i)$ Buffer = A $\}$

$l_o : \langle x := A[i] \rangle$

$\{ X = A[i] \wedge i \leq M$

$\wedge$ at $m_o \rightarrow i-j = full,\ i-j = full + 1$

$\wedge$ at $m_{o..2} \rightarrow i-j = N-(empty),\ i-j = N - empty + 1$

$\wedge$ at $m_{o..2} \rightarrow [j:i)$ Buffer = A, $(j:i)$ Buffer = A $\}$

$l_1 : \langle$wait (empty)$\rangle$

$\{ X = A[i] \wedge i \leq M$

$\wedge$ at $m_o \rightarrow i-j = full,\ i-j = full+1$

$\wedge$ at $m_{o..2} \rightarrow i-j = N-(empty+1),\ i-j = N\text{-}empty$

$\wedge$ at $m_{o..2} \rightarrow [j:i)$ Buffer=A, $(j:i)$ Buffer = A $\}$

This means buffer length = i-j, at $m_{o..2}$

also $(i-j = N-1 - empty) \supset i-j \leq N-1$, at $m_{o..2}$

so that buffer length $\leq$ N-1, at $m_{o..2}$

AND buffer length = i-j-1, at $m_{3,4}$

also $(i-j = N\text{-}empty) \supset i-j \leq N$, at $m_{3,4}$

so that again buffer length = $i-j-1 \leq N-1$, at m_3, m_4

$l_2 : \langle$Buffer $[i \bmod N] := x \rangle$

$\{$Buffer $[i \bmod N] = A[i] \wedge i \leq M$

$\wedge$ at $m_o \rightarrow i-j = \text{full}, i-j = \text{full} + 1$

$\wedge$ at $m_{o..2} \rightarrow i-j = N - (\text{empty}+1), i-j = N-\text{empty}$

$\wedge$ at $m_{o..2}$ $[j:i]$ Buffer $= A$, $(j:i]$ Buffer $= A\}$

l_3 : $\langle$signal (full)$\rangle$

$\{i \leq M$

$\wedge$ at $m_o \rightarrow i-j = \text{full}-1, i-j = \text{full}$

$\wedge$ at $m_{o..2} \rightarrow i-j = N-(\text{empty}+1), i-j = N-\text{empty}$

$\wedge$ at $m_{o..2} \rightarrow [j:i]$ Buffer=A, $(j:i]$ Buffer $= A\}$

l_4 : $\langle i := i+1\rangle$

$\{(i \leq M \vee i = M+1)$

$\wedge$ at $m_o \rightarrow i-j = \text{full}, i-j = \text{full}+1$

$\wedge$ at $m_{o..2} \rightarrow i-j = N - \text{empty}, i-j = N+1 - \text{empty}$

$\wedge$ at $m_{o..2} \rightarrow [j:i)$ Buffer $= A$, $(j:i)$ Buffer $= A\}$

Consumer: $\{j \leq M \wedge I1$

$\wedge$ at $l_{o..3} \rightarrow i-j = \text{Full}, i-j = \text{Full}-1$

$\wedge$ at $l_{o,1} \rightarrow i-j = N-\text{empty}, i-j = N - (\text{empty} + 1)$

$\wedge$ at $l_{o..2} \rightarrow [j:i)$ Buffer $= A$, $[j:i]$ Buffer $= A\}$

m_o : $\langle$wait (full)$\rangle$

$\{j \leq M \wedge I1$

$\wedge$ at $l_{o..3} \rightarrow i-j = \text{full}+1, i-j = \text{Full}$

$\wedge$ at $l_{o,1} \rightarrow i-j = N-\text{empty}, i-j = N - (\text{empty} + 1)$

$\wedge$ at $l_{o..2} \rightarrow [j:i)$ Buffer $= A$, $[j:i]$ Buffer $= A\}$

Consider at $l_{o..2}$, then

$$\text{buffer length} = i-j$$

$$\text{also}(i-j = \text{full} + 1) \supset i-j \geqslant 1$$

$$\text{hence buffer length} \geqslant 1 \text{ at } l_{o..2}$$

now suppose at $l_{3,4}$ then

$$\text{buffer length} = i+1 - j$$

also,

$$(i-j \geqslant \text{Full}) \supset$$

$$(i-j \geqslant 0) \supset (i-j + 1 \geqslant 1)$$

$$\text{hence buffer length} \geqslant 1 \text{ at } l_{3,4}.$$

$$m_1 : \langle y := \text{Buffer } [j \bmod N] \rangle$$

$$\{ y = \text{Buffer}[j \bmod N] = A[j] \wedge j \leqslant M \wedge I1$$

$$\wedge \text{ at } l_{o..3} \rightarrow i-j = \text{full}+1,\ i-j = \text{full}$$

$$\wedge \text{ at } l_{o,1} \rightarrow i-j = N\text{-empty},\ i-j = N-(\text{empty}+1)$$

$$\wedge \text{ at } l_{o..2} \rightarrow [j:i) \text{ Buffer} = A, [j:i] \text{ Buffer} = A \}$$

$$m_2 : \langle \text{signal (empty)} \rangle$$

$$\{ y = A[j] \quad \wedge I1 \wedge j \leqslant M$$

$$\wedge \text{ at } l_{o..3} \rightarrow i-j = \text{full} + 1,\ i-j = \text{full}$$

$$\wedge \text{ at } l_{o,1} \rightarrow i-j = N+1 - \text{empty},\ i-j = N - \text{empty}$$

$$\wedge \text{ at } l_{o,2} \rightarrow (j:i) \text{ Buffer} = A,\ (j:i] \text{ Buffer} = A \}$$

The Buffer assertion has to be weakened because the Buffer is circular - so that after signal (empty), possibly $A[j] \neq$ Buffer $[j \bmod N]$ but instead $A[j + N] =$ Buffer $[j \bmod N]$, i.e. the producer puts a fresh element $A[j+N]$ into empty slot Buffer $[j \bmod N]$.

$m_3 : \langle B[j] : = y \rangle$

$\{ B[j] = A[j] \wedge j \leq M \wedge I1$

$\wedge \; at\, l_{0..3} \rightarrow i - j = full + 1,\; i - j = full$

$\wedge \; at\, l_{0,1} \rightarrow i - j = N + 1 - empty,\; i - j = N - empty$

$\wedge \; at\, l_{0..2} \rightarrow (j:i)\ Buffer = A,\ (j:i]\ Buffer = A \}$

$m_4 : \langle j : = j + 1 \rangle$

$\{ (j \leq M \vee j = M+1) \wedge I1$

$\wedge \; at\, l_{0..3} \rightarrow i - j = full,\; i - j = full - 1$

$\wedge \; at\, l_{0,1} \rightarrow i - j = N - empty,\; i - j = N - (empty + 1)$

$\wedge \; at\, l_{0..2} \rightarrow [j:i)\ Buffer = A,\ [j:i]\ Buffer = A \}$

Having obtained the relevant formulas for the atomic actions, the formulas for composite actions can be derived, using the inference rules for sequencing and 'while' and finally that for cobegin.

Cobegin

Producer: $\{P_{prod}\}$ while $\langle i \leq M \rangle$ do

$\{P_{prod} \wedge i \leq M\}$

S_{prod}

$\{P_{prod}\}$

od

$\{P_{prod} \wedge i > M\}$

Here S_{prod} is the sequence of $l_0; l_1; l_2; l_3; l_4$

||

Consumer: $\{P_{cons}\}$ while $\langle j \leq M \rangle$ do

$\{P_{cons} \wedge j \leq M\}$

S_{cons}

$\{P_{cons}\}$

od

Here S_{cons} is the sequence of $m_0; m_1; m_2; m_3; m_4$

$\{P_{cons} \wedge j > M\}$

Coend.

By acanning the atomic action formulas of the Producer and Consumer, P_{prod} and P_{cons} are seen to be

$$P_{prod} \equiv \{ at\ l_{0..3} \rightarrow (at\ m_0 \rightarrow i-j=full, i-j=full+1), (at\ m_0 \rightarrow i-j=full-1, i-j=full)$$

$$\wedge at\ l_{0,1} \rightarrow (at\ m_{0..2} \rightarrow i-j=N-empty, i-j=N+1-empty),$$
$$(at\ m_{0..2} \rightarrow i-j=N-(empty+1),\ i-j = N-empty)$$

$$\wedge at\ l_{0..2} \rightarrow (at\ m_{0..2} \rightarrow [j:i)\ Buffer=A,\ (j:i)\ Buffer=A),$$
$$(at\ m_{0..2} \rightarrow [j:i]\ Buffer = A,\ (j:i]\ Buffer = A)$$

$$\wedge at\ l_{1,2} \supset X = A[i] \}$$

$$P_{cons} \equiv \{ I1$$
$$\wedge at\ m_0 \rightarrow (at\ l_{0..3} \rightarrow i-j=full, i-j=full-1), (at\ l_{0..3} \rightarrow i-j=full+1$$
$$i-j=full)$$

$$\wedge at\ m_{0..2} \rightarrow (at\ l_{0,1} \rightarrow i-j=N-empty; i-j=N-(empty+1)),$$
$$(at\ l_{0,1} \rightarrow i-j = N+1-empty,\ i-j=N-empty)$$

$$\wedge at\ m_{0..2} \rightarrow (at\ l_{0..2} \rightarrow [j:i)\ Buffer=A, [j:i]\ Buffer =A),$$
$$(at\ l_{0..2} \rightarrow (j:i)\ Buffer=A,\ (j:i]\ Buffer =A)$$

$$\wedge at\ m_{2,3} \supset y = A[j]$$

$$\wedge at\ m_4 \supset B[j]= A[j] \}$$

Finally using the cobegin rule of inference,

$\{P\}$

cobegin

Producer: $\{P\}$ while $\langle i \leq M \rangle$ do

S_{prod}

od

$\{P\}$

$\|$

Consumer: $\{P\}$ while $\langle j \leqslant M \rangle$ do

S_{cons}

od

$\{P\}$

Coend

$\{P\}$

From P_{prod}, P_{cons}, it is seen that

$P \equiv \{$ at $l_{0..3} \rightarrow$ (at $m_0 \rightarrow$ i-j=full, i-j=full+1), (at $m_0 \rightarrow$ i-j=full-1, i-j = full)

$\wedge$ at $l_{0,1} \rightarrow$ (at $m_{0..2} \rightarrow$ i-j=N-empty, i-j=N+1-empty),
(at $m_{0..2} \rightarrow$ i-j=N-(empty+1), i-j=N-empty)

$\wedge$ at $l_{0..2} \rightarrow$ (at $m_{0..2} \rightarrow$ [j:i) Buffer=A, (j:i) Buffer=A),
(at $m_{0..2} \rightarrow$ [j:i] Buffer=A, (j:i] Buffer = A)

$\wedge$ I_1

$\wedge$ at $l_{1,2} \supset x = A[i]$

$\wedge$ at $m_{2,3} \supset y = A[j]$

$\wedge$ at $m_4 \supset B[j] = A[j]$

$\wedge$ after (Producer) $\supset i > M$

$\wedge$ after (Consumer) $\supset j > M$ $\}$

Hence at the end of the program, (P $\wedge$ after (Producer) $\wedge$ after (consumer)) $\supset$

$(I1 \wedge (j > M)) \supset$

$(A[k] = B[k], \forall k: 1 \leqslant k \leqslant M)$.

This is the required output assertion.

CHAPTER 2

METHODS FOR DERIVING LIVENESS PROPERTIES

Liveness properties state that something must happen. The most well known liveness property for sequential programs is termination. Concurrent programs may have other liveness properties of interest. Indeed, cyclic concurrent programs never terminate. Some examples are:

- A process progresses through specified control points.
- Starvation Freeness: Every request for a non-shareable resource is granted .
- If an unbounded number of messages is input to a communication medium, some message is output .

A state of a program is determined by the values of all its variables and the control locations (program counter values) of all its constituent processes.

A liveness property expresses progression between one family of states and another family of states. Each of this pair of families-of-states is characterised by an assertion.

For a given program, a pair of assertions (P,Q) is in the relation $\leadsto$ (leadsto), precisely when the program is guaranteed to reach a state satisfying Q, starting from any state satisfying P.

(P,Q) are in the relation $\leadsto$ is also written as $P \leadsto Q$. The relation $\leadsto$ has two useful properties:

(i) Transitivity. If $P \leadsto R$ and $R \leadsto Q$ then $P \leadsto Q$.
This allows the task of deriving $P \leadsto Q$ to be broken into a series of steps.

(ii) Set Inductivity. If S_p is a set of assertions and $P \rightsquigarrow Q$ for all $P \in S_p$, then $(\bigvee S_p) \rightsquigarrow Q$.

($\bigvee S_p$ is obtained by ORing all assertions in S_p).

This property is used for induction on a well founded set of assertions.

A set S_p with an irreflexive partial ordering '$<$', is well founded if

for all $P \in S_p$, the set $P_< = \{x \mid x \in S_p \wedge x < P\}$ is a finite set.

For a well founded set of assertions S_p, with the above property,

If $P \rightsquigarrow [Q \vee (\bigvee P_<)]$ for all $P \in S_p$ then $(\bigvee S_p) \rightsquigarrow Q$.

For some programs, the above rule is used to derive liveness properties by induction on either

(i) some function of the values of variables, or

(ii) some function of the control locations of all constituent processes.

In contrast to safety proofs, which require local reasoning and an exhaustive checking of all elementary actions, a liveness proof is often complex and may be subtle.

Owicki-Lamport [OL] and Manna-Pnueli [MP] both use temporal logic to state liveness properties, and give axioms and rules to derive liveness properties for program fragments.

A program in [OL] is coded in a simple programming language with assignment, sequencing, while statement, cobegin statement and (as an extension) semaphores. In [MP] a program is represented as a directed graph, whose nodes are control locations, and arcs are elementary (atomic) actions. Lamport [LAM1] represents programs by flowcharts, i.e. a directed graph

with nodes as elementary actions and arcs indicating control flow (an arc may be taken to be a control location). [LAM1], being an early work, does not use temporal logic. Two axioms about the relation $\leadsto$ are given, and four theorems. All liveness properties are derived using this, in [LAM1].

2.1 Owicki-Lamport Method

In [OL] $P \leadsto Q$ is represented by the temporal logic formula $\Box(P \supset \nabla Q)$.

The two axioms are:

Atomic actions always terminate.

Atomic assignment axiom. For any atomic assignment Statement S,

$$\text{at } S \leadsto \text{after } S.$$

While Control Flow Axiom. For the statement W: While b do S od,

$$\text{at } W \leadsto (\text{at } s \vee \text{after } w).$$

There are two additional axioms for the P and V operations on semaphores (in the extended language).

V operations Axioms. For the statement $1: \langle V\ (s)\rangle$

Safety $\{Q^{s}_{S+1}\}\langle V(s)\rangle \quad \{Q\}$

Liveness at $1 \leadsto$ after 1

P operation Axioms. For the statement $1: \langle P(s)\rangle$

Safety $\{Q^{s}_{S-1}\}\langle P(s)\rangle \quad \{Q \wedge s > 0\}$

Liveness $(\text{at } 1 \wedge \Box\nabla(s > 0)) \leadsto$ after 1.

Additional rules are derived from the above two liveness axioms and various safety properties.

Concatenation Control Flow. For the statement, S;T

$$\frac{\text{at } S \leadsto \text{after } S,\ \text{at } T \leadsto \text{after } T}{\text{at } S \leadsto \text{after } T}$$

Cobegin control flow. For the statement, c:cobegin S || T coend

$$\frac{\text{at } S \leadsto \text{after } S,\ \text{at } T \leadsto \text{after } T}{\text{at } c \leadsto \text{after } c}$$

Single Exit Rule. For any statement s

$$\text{in } s \supset \Box\, \text{in } s \vee \Diamond\, \text{after } s$$

Atomic Statement Rule. For any atomic statement S

$$\frac{\{P\}\ \langle s\rangle\ \{Q\},\ \Box(\text{at } S \supset P)}{\text{at } S \leadsto (\text{after } S \wedge Q)}$$

General Statement Rule. For any statement S

$$\frac{\{P\}\ S\ \{Q\},\ \Box(\text{in } S \supset P),\ \text{in } S \leadsto \text{after } S}{\text{in } S \leadsto (\text{after } S \wedge Q)}$$

While Test Rule. For the statement W:While $\langle b\rangle$ do S od

$$\text{at } W \leadsto ((\text{at } S \wedge b) \vee (\text{after } W \wedge \neg b))$$

While Exit Rule. For the statement W: While $\langle b\rangle$ do S od

$$\text{at } W \wedge \Box(\text{at } W \supset b) \leadsto \text{at } S$$

$$\text{at } W \wedge \Box(\text{at } W \supset \neg b) \leadsto \text{after } W$$

Liveness properties are derived from a proof lattice. A proof lattice has a structure which allows rigorous, but compact and high-level derivations to be made. The level of reasoning, being at a higher level than the individual steps of a fully formal proof, aids in comprehension.

A proof lattice is defined to be a finite directed acyclic graph, in which each node is labelled with an assertion, such that

(i) There is a single entry node having no incoming edges.

(ii) There is a single exit node having no outgoing edges.

(iii) If a node labelled R has outgoing edges to nodes labelled R_1, $R_2, \ldots, R_k$, then

$$R \leadsto (R_1 \vee R_2 \vee \ldots \vee R_k) \text{ holds for the program.}$$

From the third condition follows, that, if R is true at some time, at least one of the R_i must be true at some later time. As a consequence, if the entry node assertion is true at some time, then the exit node assertion must be true at some later time. Hence, the theorem,

If there is a proof lattice for a program with entry node labelled P and exit node labelled Q, then $P \leadsto Q$ is true for that program.

The advantage of a proof lattice for deriving liveness properties is its ease of understanding and high-level reasoning.

The main drawback is that a restricted notation is used for derivations which may be quite complex. This sometimes leads to convoluted uses of the notation, which may be misleading.

(i) Mixup of $\supset$, $\equiv$ and $\leadsto$.

This is justified by the temporal logic theorem

$$\Box(P \supset Q) \supset P \leadsto Q \text{ .}$$

The safety property $I \supset \Box I$ may be represented as

$$\begin{array}{c} I \\ \downarrow \\ \Box I \end{array}$$

Similarly in $c \equiv$ at $c \vee$ at b, may be represented as

$$\begin{tikzcd} & \text{in c} \arrow[dl] \arrow[dr] & \\ \text{at c} & & \text{at b} \end{tikzcd}$$

(ii) A single step in the proof lattice (i.e. an edge) may require quite a complicated formal proof involving detailed examination of a sizeable fragment of the program. Of course, this is precisely the motivation for using high-level reasoning. Neverthless, each step in the proof lattice must be 'obvious'. These drawbacks can be avoided in a careful derivation.

A program for two-process mutual exclusion [PET], is taken as an example.

```
                 Program Mutex;

   c: Q1, Q2: Boolean; Last:integer;...other variables;
      Q1: = false, Q2: = false; Last: = 1;
  sc: cobegin
        Process (1) || Process (2)
      coend.
```

```
Process (1);
W1: While true do
   NC1: Noncritical Section 1;
    l1: Q1: = True;
    m1: Last: = 1;
    p1: While ⟨ Q2⟩ ∧ ⟨Last=1⟩ do
          q1: skip
          od;
   cs1: Critical Section 1;
   r1 : Q1: = False
       od
```

```
Process (2);
W2: While true do
   NC2: Noncritical Section 2;
    l2: Q2: = True;
    m2: Last: = 2;
    p2: While ⟨ Q1⟩ ∧ ⟨Last = 2⟩ do
          q2: skip
          od;
   cs2: Critical Section 2;
    r2: Q2: = False
        od
```

Process 1 and 2 are similar, hence, if Process 1 has the desired liveness property, then so does Process 2.

It is assumed that no Process (i) stays forever inside its critical section

i.e. in $CS_i \leadsto$ after cs_i.

The desired liveness property is

(L1) at $c \supset$ (at $l_1 \leadsto$ at cs_1).

The following safety properties are used,

(s1) $I \supset \Box I$,

where

$I \triangleq$ (in $sc \supset$ (Last = 1 $\vee$ Last = 2))

$\wedge$ (at $w_i \vee$ at $l_i \vee$ in $NC_i \supset \neg Q_i$), for $i = 1,2$

$\wedge$ (at $m_i \vee$ in $P_i \vee$ in $cs_i \vee$ at $r_i \supset Q_i$), for $i = 1,2$

$\wedge$ (in $cs_1 \wedge$ in $P_2 \supset$ Last=2) $\wedge$ (in $cs_2 \wedge$ in $P_1 \supset$ Last=1)

The first three terms of I are obviously invariant. Examining the fourth term, i.e. (in $cs_1 \wedge$ in $P_2 \supset$ Last = 2) the only program actions affecting it are at m_1, p_1, m_2.

m_1, however, cannot falsify the fourth term, because immediately after it is executed 'in P_1' is true, i.e. the antecedent of the term is false, since in $P_1 \supset \neg$ in cs_1.

Considering P_1, suppose it indeed did falsify the fourth term, by making in CS_1 true while Last $\neq$ 2,

$\{$in $CS_1 \wedge$ in $P_2 \supset$ Last=2$\}$ P_1:...$\{$in $CS_1 \wedge$ in $P_2 \wedge \neg$(Last=2)$\}$.

This implies that, immediately after P_1 is executed, (in $P_2 \wedge$ Last $\neq$ 2) is true, i.e. (Q2 $\wedge$ Last = 1) is true. Now as P_1 does not modify any variables, immediately before P_1 was executed, (Q2 $\wedge$ Last=1) must have been true. But, if this

CENTRAL LIBRARY
Kanpur.
Acc. No. A 82795

were the case, control would have transferred to q_1 and not cs_1. Contraction.

m_2 always makes Last = 2 and so cannot falsify the fourth term.

The fifth term may be similarly proved.

$I \supset \Box I$ and at $c \supset I$, together imply

$$\text{at } c \supset \Box I \ .$$

From the invariance of the fourth and fifth terms of I, it is easy to see the invariance of the mutual exclusion property

(S2) $\neg(\text{in } cs_1 \wedge \text{in } cs_2) \supset \Box \neg(\text{in } cs_1 \wedge \text{in } cs_2)$.

The only statements affecting 'in cs_i' are P_1, P_2. Neither P_1 nor P_2 can falsify $\neg(\text{in } cs_1 \wedge \text{in } cs_2)$.

Considering P_1, it could only falsify this assertion by making in cs_1 true, while process 2 is already in its critical section. But then, the invariance of the fifth term of I, implies that control would transfer from P_1 to q_1 (and not cs_1).

From at $c \supset \neg(\text{in } cs_1 \wedge \text{in } cs_2)$ and the invariance of $\neg(\text{in } cs_1 \wedge \text{in } cs_2)$ follows that

$$\text{at } c \supset \Box \neg(\text{in } cs_1 \wedge \text{in } cs_2).$$

Another required property is,

(S3a) $\text{Last} = 2 \wedge \Box(\text{in } P_1) \supset \Box(\text{Last} = 2)$

(S3b) $\text{Last} = 1 \wedge \Box(\text{in } P_2) \supset \Box(\text{Last} = 1)$

The property (S3a) is true, because the only program action falsifying Last = 2 is m_1, and control, being forever in P_1, could never reach m_1.

Lastly, (54a) at $W_1 \supset \Box$ in W_1

(54b) at $W_2 \supset \Box$ in W_2.

Again from at sc $\supset$ at W_1, follows

at sc $\supset \Box$ in W_1.

Similarly at sc $\supset \Box$ in W_2.

The program must reach the cobegin, when started, i.e. at c $\leadsto$ at sc.

Consequently, $(\text{at } c \supset \Box(I)) \wedge [\text{at } c \supset \nabla \Box(\text{in } W_1 \wedge \text{in } W_2)]$.

A proof lattice is used to prove $\Box I$ and $\nabla\Box(\text{in } W_1 \wedge \text{in } W_2)$ and (53a) and (53b) implies $(\text{at } l_1 \leadsto \text{at } cs_1)$.

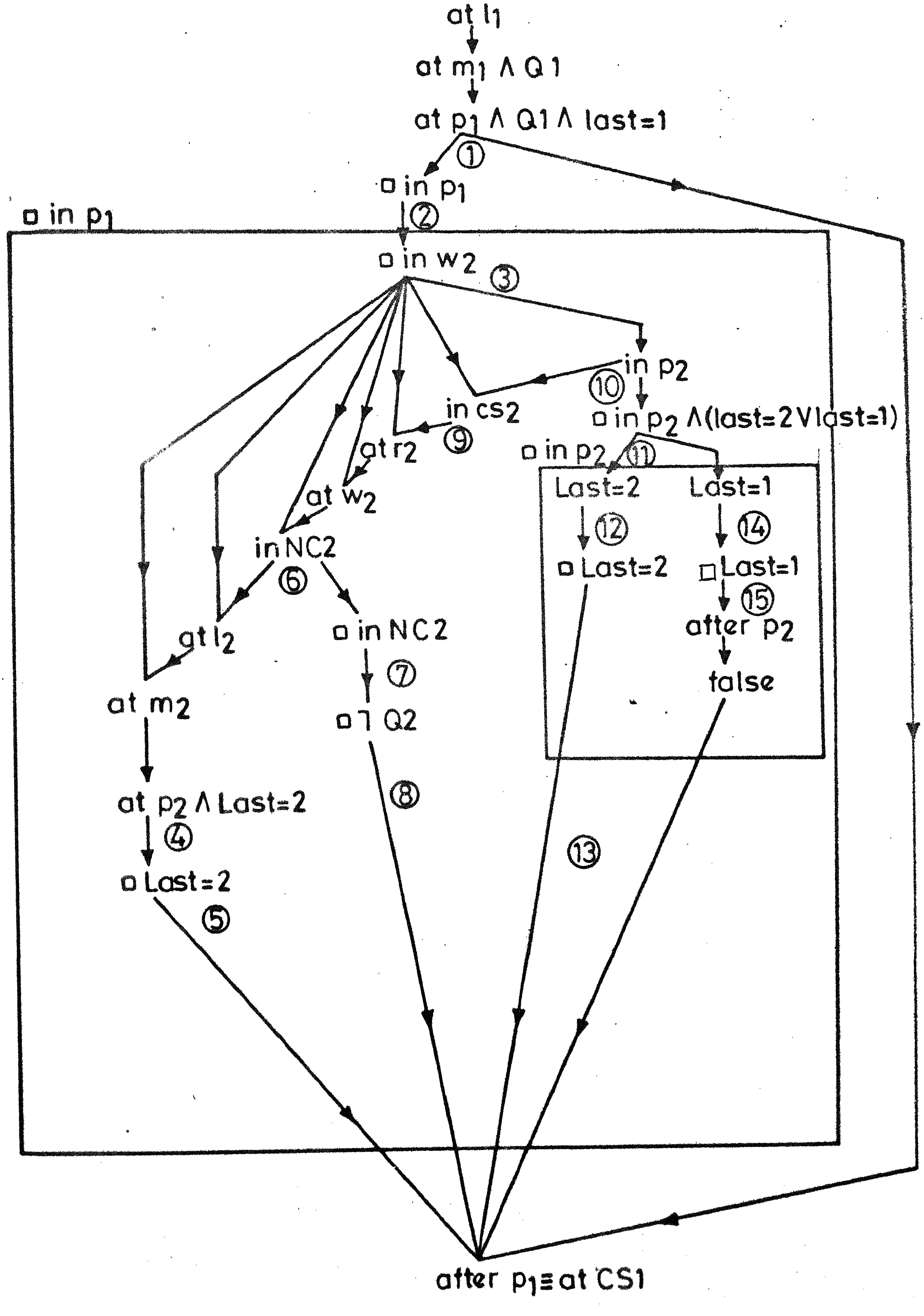
Combined with the previous step, this gives the desired liveness property,

$$\text{at } c \supset [\text{at } l_1 \leadsto \text{at } cs_1].$$

PROOF LATTICE

$\nabla\Box$(in $w_1 \wedge$ in w_2) and $\Box$(I) and (S3a) and (S3b) implies

[at $l_1 \rightsquigarrow$ at CS1]



The various steps in the proof lattice are derived as follows:

(1) Single exit rule, $\text{in } P_1 \supset \Box \text{in } P_1 \vee \nabla \text{after } P_1$

(2) $\Box$ in W_2 is assumed to hold eventually.

(3) $\text{in } W_2 \equiv (\text{at } m_2 \vee \text{at } l_2 \vee \text{in } NC_2 \vee \text{at } W_2, \phi_2 \vee \text{in } cs_2 \vee \text{in } p_2)$ and using the theorem $\Box(P \vee Q) \supset \nabla P \vee \nabla Q$

(4) For all these nodes within the outer box, $\Box$ in P_1 is true. From $\Box$in P_1, Last = 2 and (53a), follows $\Box$ Last=2.

(5) While Exit rule for P_1.

(6) Single exit rule, $\text{in } NC_2 \supset (\Box \text{in } NC_2 \vee \nabla \text{ after } NC_2)$.

(7) From $\Box$ in NC_2 and the invariant I (i.e. $\Box(\text{in } NC_2 \supset \neg Q2)$) follows $\Box \neg Q2$.

(8) While Exit Rule for P_1.

(9) By assumption in $cs_i \leadsto$ after cs_i.

(10) Single Exit Rule $\text{in } P_2 \supset (\Box \text{ in } P_2 \vee \nabla \text{ after } P_2)$.
(Last =2 $\vee$ Last=1) is assumed to hold in sc, by $\Box$ I.

(11) The theorem (Last =2 $\vee$ Last=1) $\supset \nabla$ Last=2 $\vee$ ∇ Last=1.

(12) From $\Box$ in P_1, Last=2 and (53a), follows $\Box$ Last = 2.

(13) While Exit Rule for P_1.

(14) From $\Box$ in P_2, Last=1 and (53b), follows $\Box$ Last=1.

(15) While Exit Rule for P_2.
'after P_2' contradicts the fact that $\Box$ in P_2 holds for all nodes within the inner box. Hence, the next edge leading to 'false'.

The other steps in the proof lattice follow from the Atomic Statement Rule.

*Box Notation: Drawing a box labelled $\Box Q$ around some nodes in the lattice (Q is any assertion) signifies that $\Box Q$ is to be ANDed to the assertion attached to every node in the box.

2.2 Manna-Pnueli Method

In [MP], each process (sequential) of a concurrent program is represented as a directed graph, in which the nodes are the control locations. Each arc represents an elementary action. Associated with each arc is an enabling predicate, which must be true for the elementary action to occur, and an updating function which updates the values of all variables simultaneously and also updates the location variable (i.e. program counter), of the process to which the arc belongs .

Four rules are given, to derive $P \leadsto Q$ for a given subgraph of a directed graph (i.e. for a given program fragment) $P \leadsto Q$, in MP terms, is the Temporal Logic theorem $\models P \supset \lozenge Q$.

Actually, the rules are used to derive formulas of the form $P \wedge \square \chi \supset$ (some consequent). That is, progression between P and (say) Q is guaranteed, subject to the invariance of χ, (i.e. $\square \chi \supset (P \leadsto Q)$). In those cases where no additional invariant is required to guarantee progression, $\chi \equiv$ True.

The four rules follow. MP uses '(at $l \wedge \emptyset$)' for P, and 'ψ' for Q.

A predicate $\emptyset$ is said to be χ-invariant if $\emptyset$ is preserved by every transition which preserves χ.

i.e. for every elementary action, α,

$$\{\emptyset \wedge \chi\} \quad \alpha \quad \{\emptyset \vee \neg \chi\}$$

That is, for every α,

$$\left[\emptyset(\bar{\pi}; \bar{y}) \wedge \chi(\bar{\pi}; \bar{y})\right] \supset \left[\emptyset(r_\alpha(\bar{\pi}); f_\alpha(\bar{y})) \vee \neg\chi(r_\alpha(\bar{\pi}); f_\alpha(\bar{y}))\right]$$

<u>(ESC) Rule of Escape</u>

Consider a location l in process P_j. Let $\Sigma = \{\alpha_1,\ldots,\alpha_k\}$ be some subset of the set of all transitions orginating in l. Let $l^1,\ldots,l^k$ be the destination locations and $c_1,\ldots,c_k$ be the enabling predicates of transitions $\alpha_1,\ldots,\alpha_k$. The location l must be deterministic, i.e. the enabling predicates c and c' of any two distinct transitions originating in l must be disjoint, so that $\neg(c \wedge c')$. Let $\emptyset$, χ and ψ be predicates such that:

A: $\emptyset$ is (at $l \wedge \chi$)-invariant.

This means that as long as control remains at l and χ is presered, so is $\emptyset$.

B: Any of the α_i, i=1,...,K, transitions of Σ that preserves χ and is initiated with $\emptyset$ true, achieves ψ. i.e.

$$\left(\text{at } l \wedge c_i\ (\bar{y}) \wedge \emptyset(\bar{\pi};\bar{y}) \wedge \chi(\bar{\pi};\ \bar{y}) \wedge \chi(r_i(\bar{\pi});\ f_i(\bar{y}))\right)$$
$$\supset\ \psi(r_i(\bar{\pi});\ f_i(\bar{y}))$$

for every i=1,...,K.

Here r_i is a function on the vector of location variables that updates the location variable of process P_j from l to l^i (all other location variables are maintained).

C: $\emptyset \wedge \chi \wedge$ at l ensures that at least one c_i, i=1,...,K is true (i.e. at least one transition α_i is enabled).

Then under these conditions

$$\models\left(\text{at } l \wedge \emptyset \wedge \Box\chi\right) \supset \nabla\psi$$

(ALT) Rule of Alternatives

This rule applies to a set of (possibly nondeterministic) locations. Let L be a set of locations in the process P_j and $\Sigma = \{\alpha_1, \ldots, \alpha_k\}$ the set of all transitions originating in L and leading to locations $l^1, \ldots, l^k$ outside of L, i.e. $l^i \notin L$.

Let $\emptyset, \chi, \psi$ be predicates such that:

A: $\emptyset$ is (at $L \wedge \chi$)-invariant.

This means that as long as control remains in L and χ is preserved, so is $\emptyset$.

B: Any of the α_i, i=1,...,K, transitions of Σ that preserves χ and is initiated with $\emptyset$ true, achieves ψ, i.e. ψ will hold after the transition.

Then under these conditions

$$\models (\text{at } L \wedge \emptyset \wedge \Box \chi) \supset (\Box(\text{at } L \wedge \emptyset) \vee \nabla \psi)$$

(SEM) Semaphore Rule

Rule ESC above is adequate for dealing with locations for which the disjunction of all their enabling predicates (on all the outgoing transitions) is identically true. A location which does not satisfy this requirement is called a semaphore location. A stronger rule than ESC and ALT, is required to reason about semaphore locations.

Let l be a (possibly semaphore) location and $\Sigma = \{\alpha_1, \ldots,$ the set of all the transitions originating in l. Let l^i and c^i, for i=1,...,K, be respectively the destination location of α_i and its enabling predicate.

Let ϕ, χ and ψ be predicates such that:

A: ϕ is (at $l \wedge \chi$)-invariant.

This means that as long as control remains at l and χ is preserved, so is ϕ.

B: Any of the α_i, $i=1,\ldots,K$, transitions of Σ, which preserves χ and is initiated with ϕ true, achieves ψ.

C: If $(\phi \wedge \chi)$ hold permanently at l, then eventually one of the c_i, $i=1,\ldots,K$, will be true. That is

$$\models \Box(\text{at } l \wedge \phi \wedge \chi) \supset \nabla \bigvee_{i=1}^{k} c_i.$$

Under these conditions

$$\models (\text{at } l \wedge \phi \wedge \Box\chi) \supset \nabla\psi$$

(SP) Single Path Rule

In this derived rule, ESC is applied repetitively to a sequence of locations.

Let $l^1, l^2, \ldots, l^{k+1}$ be a path of deterministic locations in P_j with an immediate transition α_i from every l_i to l_{i+1}, $i=1,\ldots,k$.

$$l_1 \xrightarrow{c_1(\bar{y}) \to [\bar{y} := f_1(\bar{y})]} l_2 \longrightarrow \cdots\; l_k \xrightarrow{c_k(\bar{y}) \longrightarrow [\bar{y} := f_k(\bar{y})]} l_{k+1}$$

Let χ, $\phi_1, \ldots, \phi_k$ and $\phi_{k+1} = \psi$ be predicates such that:

A: Each ϕ_i is (at $l_i \wedge \chi$)-invariant, $i=1,\ldots,K$.

This means that as long as control remains at l_i and χ is preserved, so is ϕ_i.

B: Each transition α_i, $i=1,\ldots,K$, which preserves χ and is initiated with ϕ_i true, achieves ϕ_{i+1}.

C: $(\emptyset_i \wedge \chi)$ at l_i ensures that c_i is true, i.e.

$$[\text{at } l_i \wedge \emptyset_i \wedge \chi] \supset c_i.$$

Then under these conditions

$$\models \left(\bigvee_{i=1}^{k} (\text{at } l_i \wedge \emptyset_i) \wedge \Box \chi \right) \supset \nabla \psi.$$

That is, if control is anywhere in the path with the appropriate $\emptyset_i$ true and χ is continuously maintained, eventually ψ is true.

2.2.1 Example-Generalized Dining Philosophers

A number of philosophers are seated round a table, on which there are the same number of forks. A philosopher continuously cycles between Thinking, picking up the two forks on either side and Eating, then again Thinking,.....[DIJ1] .

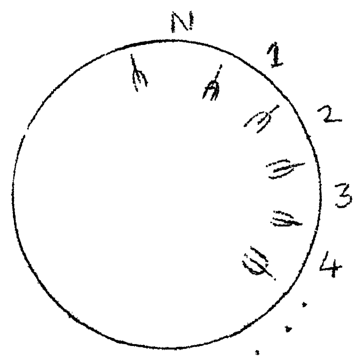
It is assumed that the number of philosophers is at least three.

```
Program Dining Philosophers;
{ 3 ≤ N = some constant }
Mutex:  Semaphore;
Privsem: Array [1..N] of semaphore; State:Array [1..N] of
         integer;
Mutex: = 1;
For all i, 1 ≤ i ≤ N  do
    State [i]:=0 ; Privsem [i]:= 0
                od;
    cobegin
    Philosopher (1) || Philosopher (2) || ....|| Philosopher (N)
    coend.
```

```
Philosopher (i)
Th: Think;
k  : Wait (mutex);
l  : State [i]: = 1;
m  : Test & Set (i);
p  : Signal (mutex);
q  : Wait (Privsem [i]);
Eat: The philosopher eats;
r  : Wait (mutex);
s  : State [i]:=0;
u  : Test & Set (Li);
v  : Test & Set (Ri);
w  : Signal(mutex);
y  : Go to Th;
```

Test & Set (i) is an abbreviation for the atomic action:

$$\langle \text{If State}[Li] \neq 2 \;\wedge \text{State}[i] = 1 \;\wedge \text{State}[Ri] \neq 2 \text{ then State}[i] := 2, \text{ Signal (Privsem (i))} \rangle$$



Note: State may be interpreted as,

State [i]= 0 - Philosopher (i) is Thinking;

State [i]= 1 - Philosopher (i) is Hungry;

State [i]= 2 - Philosopher (i) is Eating.

Li and Ri are the left and right neighbour of philosopher i, respectively, Thus for philosophers numbered 1 to N,

L(i) $\equiv$ If i=N then 1 else i+1,

R(i) $\equiv$ If i=1 then N else i-1.

The control location of each philosopher is shown by the subscripted label of the control location, where the subscript is the number of the philosopher.

Thus (at Th_{Li}) is true if the left neighbour of philosopher i is thinking. The subscript is dropped in case of philosopher i, itself. Thus (at Eat) is true if philosopher i is eating.

Subject to a constraint, it is proved that for any arbitrary philosopher i,

$$\models \text{at } k \supset \nabla \text{ at Eat.}$$

It is possible for the two neighbours of philosopher i, to block i from ever eating, by suitably interleaving their own eating. The following constraint prevents this:

$$\Box \text{ State}[i]= 1 \supset \nabla \neg(\text{State}[Li] = 2 \vee \text{State}[Ri] = 2)$$

Let NbrsIn $\triangleq$ (State [Li]= 2 $\vee$ State [Ri]= 2).

- Either neighbour of Philosopher (i) is Eating.

The constraint is

$$\models \Box \text{ State}[i] = 1 \supset \nabla \neg \text{NbrsIn}$$

- If a Philosopher is forever hungry, then sometime both its neighbours will not be eating.

Required Safety Properties:

Q1: $\text{Mutex} + \forall_i:(\text{at } l_i + \text{at } m_i + \text{at } p_i) + \forall_i:(\text{at } s_i + \text{at } u_i + \text{at } v_i + \text{at } w_i) = 1.$

Q2: $\forall_i : (\text{State}[i]= 0 \vee \text{State}[i]= 1 \supset \text{Privsem}[i]= 0) \wedge (\text{Mutex} \geq 0)$
$\wedge (\text{Privsem}[i]= 1 \supset \text{State}[i]= 2) \wedge \neg(\text{State}[i]= 2 \wedge ((\text{State}[Li]= 2)$
$\vee (\text{State}[Ri] = 2))$
$\wedge (\text{at } q_i \wedge \text{State}[i]= 2 \supset \text{at } q_i \wedge \text{Privsem}[i]= 1)$

Initial Condition $\triangleq$ $(\text{Mutex}=1) \wedge \forall_i:(\text{State}[i]= 0 \wedge \text{Privsem}[i]=0)$
$\wedge \forall_i:(\text{at } Th_i).$

From, Initial Condition $\supset Q1 \wedge Q2$ and the invariance of $Q1 \wedge Q2$, follow

$$\models \Box (Q1 \wedge Q2) .$$

Two other safety properties are required, which are of the form $I \wedge \Box R \supset \Box I$

S1: $(\text{at } q \wedge \text{State}[i]= 1) \wedge \Box \neg (\text{at } q \wedge \text{State}[i]= 2) \supset$
$\Box (\text{at } q \wedge \text{State}[i]= 1)$, for all i.

This states that if philosopher i is waiting on its Privsem, and neither neighbour of i does Test-and-Set (i) successfully, then philosopher i must wait forever.

The only actions affecting this implication are those at q_i, u_{Ri}, v_{Li} because only these can falsify (at $q \wedge$ State $[i]=1$). The action at q_i can never occur, as long as State $[i]= 1$. The actions at u_{Ri}, v_{Li} either fail and do nothing, or succeed and set State $[i]= 2$. But if they do succeed, the antecedent term $\Box\neg(\text{at } q \wedge \text{State}[i]= 2)$ is falsified. So that,

$$\models (\text{at } q \wedge \text{State}[i]= 1) \wedge \Box \neg (\text{at } q \wedge \text{State}[i]= 2) \supset \Box (\text{at } q \wedge \text{State}[i] = 1).$$

Let χ be $\triangleq$ $(\text{at } s_{Li} \supset \text{State}[Ri] = 2)$
$\wedge (\text{at } s_{Ri} \supset \text{State}[Li] = 2)$.

S2: $\text{NbrsIn} \wedge \Box \chi \supset \Box \text{NbrsIn}$, for all i.

In detail, this is

$$(\text{State}[Li]= 2 \vee \text{State}[Ri]= 2) \wedge \Box ((\text{at } s_{Li} \supset \text{State}[Ri]= 2) \wedge (\text{at } s_{Ri} \supset \text{State}[Li]= 2)) \supset \Box (\text{State}[Li] = 2 \vee \text{State}[Ri]= 2).$$

The only actions affecting this implication are those at s_{Li}, s_{Ri}, because for any process i the transition (State [i] = 2) $\longrightarrow$ (State [i] $\neq$ 2) is made only at a single location, i.e. S, and only such an action could falsify NbrsIn.

However, it is obvious that, if (State [Li] $\neq$ 2 $\wedge$ State [Ri] $\neq$ 2) is true immediately after either s_{Li} or s_{Ri}, then χ could not have been true immediately before the action.

Consider s_{Li},

at $s_{Li} \wedge \chi \supset$ State [Ri] = 2,

so that even after s_{Li}, State [Ri] would still remain 2.

Hence if $\Box\chi$ does indeed hold, then even the actions at s_{Li}, s_{Ri} cannot falsify NbrsIn.

The following lemmas are needed for the final proof.

Lemma 1.

$\models$ at $k_i \supset \nabla$ at l_i, for all i.

This states that no philosopher is blocked forever waiting on Mutex.

Lemma 2.

$\models$ at $q_i \wedge$ State [i] = 2 $\supset \nabla$ at Eat_i, for all i.

Any philosopher waiting on its Privsem with its State = 2, must have its Privsem = 1, and so cannot block at q.

Lemma 3.

$\models$ at $l_i \wedge$ NbrsIn $\supset \nabla$(at $q_i \wedge$ NbrsIn $\wedge$ State [i] = 1),

for all i.

Lemma 4.

$\models$ at $l_i \wedge \neg$NbrsIn $\supset \nabla$ (at $q_i \wedge$ State [i] = 2), for all i.

$\models$ (at $l_j \ldots p_j$) $\wedge$ mutex = 0 $\supset \nabla$(at $q_j \wedge$ mutex = 1), by the Single Path rule applied to path $l_j \rightarrow m_j \rightarrow p_j \rightarrow q_j$.

$\models$ (at $s_j \ldots w_j$) $\wedge$ mutex = 0 $\supset \nabla$(at $y_j \wedge$ mutex = 1), by the Single Path rule applied to $s_j \rightarrow u_j \rightarrow v_j \rightarrow w_j \rightarrow y_j$.

(2) $\models$ at k $\wedge$ mutex = 0 $\supset \nabla$ mutex = 1, by the last three steps.

$\models$ at k $\supset \nabla$ mutex = 1, by 1,2.

$\models$ at k $\supset \nabla$ at l, by Semaphore rule.

Lemma 2. Prove at q $\wedge$ State [i] = 2 $\supset \nabla$ at Eat.

Again, by Semaphore rule, it must be shown that

$\models \Box$ (at q $\wedge$ State [i] = 2) $\supset \nabla$(Privsem [i] = 1).

$\models$ at q $\wedge$ State [i] = 2 $\supset$ Privsem [i] = 1, by Q2.

$\models$ at q $\wedge$ State [i] = 2 $\supset \nabla$ at Eat, by Semaphore rule.

State [i] = 2 is (at q)-invariant, is seen to be true.

Lemma 3. Prove at l $\wedge$ NbrsIn $\supset \nabla$(at q $\wedge$ NbrsIn $\wedge$ State [i] = 1).

$\models$ at l $\wedge$ NbrsIn $\supset \nabla$ (at m $\wedge$ Nbrs In $\wedge$ State [i] = 1), by Escape rule. NbrsIn is (at l)-invariant.

$\models$ at m $\wedge$ NbrsIn $\wedge$ State [i] = 1 $\supset \nabla$(at q $\wedge$ Nbrs in $\wedge$ State [i] = 1),

by Single Path rule applied to path m $\longrightarrow$ p $\longrightarrow$ q.

(NbrsIn $\wedge$ State [i] = 1) is at m, at p-invariant.

$\models$ at l $\wedge$ NbrsIn $\supset \nabla$(at q $\wedge$ NbrsIn $\wedge$ State [i] = 1), by the last two steps.

Lemma 4. Prove at l $\wedge \neg$NbrsIn $\supset \nabla$(at q $\wedge$ State [i] = 2).

$\models$ at l $\wedge \neg$NbrsIn $\supset \nabla$ (at m $\wedge \neg$NbrsIn $\wedge$ State [i] = 1), by Escape rule.

$\models$ at m $\wedge \neg$NbrsIn $\wedge$ State [i] = 1 $\supset \nabla$(at p $\wedge$ State [i]= 2),

by Escape rule.

($\neg$ NbrsIn $\wedge$ State [i]= 1) is (at m)-invariant. Under these conditions, Test-and-Set (i) succeeds.

$\models$ at p $\wedge$ State [i]= 2 $\supset \nabla$(at q $\wedge$ State [i]= 2),

$\models$ at l $\wedge \neg$NbrsIn $\supset \nabla$ (at q $\wedge$ State [i]= 2), by the last three steps.

Lemma 5. Prove at s_{Li} $\wedge$ at q $\wedge$ State [i]= 1 $\wedge$ State [Ri]$\neq$2 $\supset \nabla$ at Eat.

$\models$ at s_{Li} $\wedge$ (at q $\wedge$ State [i]= 1 $\wedge$ State [Ri]$\neq$2) $\supset$

∇ (at v_{Li} $\wedge$ at q $\wedge$ State [i]= 1 $\wedge$ State [Ri]$\neq$ 2),

by Single Path rule applied to $s_{Li} \longrightarrow u_{Li} \longrightarrow v_{Li}$.

(at q $\wedge$ State [i]= 1 $\wedge$ State [Ri]$\neq$ 2) is (at s_{Li}, at u_{Li})-invariant.

$\models$ at v_{Li} $\wedge$ at q $\wedge$ State [i]= 1 $\wedge$ State [Ri]$\neq$ 2 $\supset$

∇ (at w_{Li} $\wedge$ at q $\wedge$ State [i]= 2), by Escape rule.

(at q $\wedge$ State [i]= 1 $\wedge$ State [Ri]$\neq$ 2) is (at v_{Li})-invariant.

Under these conditions, Test-&-Set (i) is successful.

$\models$ at q $\wedge$ State [i]= 2 $\supset \nabla$ at Eat by Lemma 2.

$\models$ at s_{Li} $\wedge$ at q $\wedge$ State[i] = 1 $\wedge$ State [Ri] $\neq$ 2 $\supset$

∇ at Eat, by last three steps.

Proof of the main liveness property.

Recall the abbreviations,

NbrsIn $\triangleq$ (State [Li] = 2 $\vee$ State [Ri] = 2).

χ .. $\triangleq$ (at s_{Li} $\supset$ State [Ri]= 2)

$\wedge$ (at s_{Ri} $\supset$ State [Li]= 2).

The assumed constraint is

(C) $\Box(\text{State}[i] = 1) \supset \nabla \neg \text{NbrsIn}.$

Prove,

$$\text{at } k \supset \nabla \text{ at Eat}.$$

(1) $\models$ at k $\supset \nabla$ at 1, Lemma 1.

(2) $\models$ at 1 $\supset$ (at 1 $\wedge$ NbrsIn) $\vee$ (at 1 $\wedge \neg$NbrsIn),

Tautology.

(3) $\models$ at 1 $\wedge \neg$ NbrsIn $\supset \nabla$ (at q $\wedge$ State [i] = 2), Lemma 4.

(4) $\models$ at 1 $\wedge \neg$ NbrsIn $\supset \nabla$ at Eat, by 3, Lemma 2.

(5) $\models$ at 1 $\wedge$ NbrsIn $\supset \nabla$ (at q $\wedge$ NbrsIn $\wedge$ State [i]=1),

Lemma 3.

(6) $\models$ at q $\wedge$ State [i]= 1 $\supset \Box$(at q $\wedge$ State [i]= 1)

$\vee \nabla$ (at q $\wedge$ State [i]= 2),

by Safety Property S1.

(7) $\models$ at q $\wedge$ State [i]= 1 $\supset \Box$(at q $\wedge$ State [i] = 1),

$\vee \nabla$ (at Eat), by 6. Lemma 2.

(8) $\models$ at q $\wedge$ State [i]= 1 $\wedge$ NbrsIn $\supset$ NbrsIn $\wedge$

($\Box$ (at q $\wedge$ State [i]= 1) $\vee \nabla$ (at Eat)),

by , ANDing (NbrsIn) to the antecedent

and the consequent, of (7).

(9) $\models$ NbrsIn $\wedge \Box$ (at q $\wedge$ State [i]= 1) $\supset \nabla \neg$ NbrsIn,

by assumption C.

(10) $\models$ NbrsIn $\wedge \Box \mathcal{X} \supset \Box$ NbrsIn, by Safety Property S2.

(11) $\models \nabla \neg$NbrsIn $\supset \neg$ NbrsIn $\vee \nabla \neg \mathcal{X}$, by 10.

(12) $\models$ NbrsIn $\wedge$ $\Box$ (at q $\wedge$ State [i]= 1) $\supset$ $\neg$NbrsIn $\vee \nabla \neg \chi$,

by 9,11.

(13) $\models$ NbrsIn $\wedge$ $\Box$ (at q $\wedge$ State [i] = 1) $\supset$

∇ (at s_{Li} $\wedge$ State [Ri] $\neq$ 2)

$\vee$ ∇ (at s_{Ri} $\wedge$ State [Li] $\neq$ 2),

by 12, rewriting $\neg\chi$.

(14) $\models$ NbrsIn $\wedge$ $\Box$ (at q $\wedge$ State [i] = 1) $\supset$

∇ (at q $\wedge$ State [i]= 1 $\wedge$at s_{Li}

$\wedge$State [Ri] $\neq$ 2)

$\vee$ ∇ (at q $\wedge$ State [i]= 1 $\wedge$at s_{Ri}

$\wedge$ State Li $\neq$ 2) by 13, Temporal

Logic Theorem*.

The theorem used is $\vdash (R \wedge \Box P \supset \nabla Q) \equiv (R \wedge \Box P \supset \nabla(P \wedge Q))$

(15) $\models$ at s_{Li} $\wedge$ at q $\wedge$ State [i] = 1 $\wedge$State [Ri] $\neq$ 2 $\supset$

∇ at Eat, by Lemma 5.

(16) $\models$ at s_{Ri} $\wedge$ at q $\wedge$ State [i] = 1 $\wedge$ State [Li] $\neq$ 2$\supset$

∇ at Eat, by Lemma 6.

(17) $\models$NbrsIn $\wedge$$\Box$(at q $\wedge$ State [i]= 1) $\supset$ ∇ at Eat,

by 14,15,16.

(18) $\models$ NbrsIn $\wedge$ at q $\wedge$ State[i] = 1 $\supset$ ∇ at Eat,

by 8,17.

(19) $\models$ at 1 $\wedge$ NbrsIn $\supset$ ∇ at Eat, by 5,18.

(20) $\models$ at 1 $\supset$ ∇ at Eat, by 2,4,19.

(21) $\models$at k $\supset$ ∇ at Eat, by 1,20.

This completes the proof.

2.3 Lamport's Method

The approach of [LAM1] is different from that of [OL], [MP]. Being an early work it does not make use of temporal logic. Each process, Pr_k, of a program is represented in [LAM1] as a flowchart. Associated with each process is a token, which is initially placed on a distinguished arc of the corresponding flowchart. Any flowchart has two kinds of nodes (i.e. elementary actions) - assignment nodes and decision nodes. The execution of a process is represented by the movement of its token from arc to arc. When the token passes through an assignment node, the values of variables are updated, whereas, if it passes through a decision node, the condition is evaluated and the token moves to the appropriate (T or F) output arc of the node.

In order to derive safety or liveness properties of a program, every arc α, of every process Pr_k of the program is annotated with a pair of assertions, called an input assertion ($_i I^k_\alpha$) and an output assertion ($_o I^k_\alpha$). (An assertion is a truth valued function of variables and token positions. The output assertion, in fact, is alway either identical to the corresponding input assertion, or it is the assertion 'false'). The idea is that whenever a token reaches an arc, the corresponding output assertion is true. In the initial state each token is on an arc whose input assertion is true. Consistency Condition (for an isolated process). For each flowchart node, if the token is on an input arc whose input assertion is true, then executing the node moves the token to an output arc of the node whose output

assertion is true.

This condition, of course, only guarantees the partial correctness of an isolated process.

The entire program is consistent if (i) each process in isolation is consistent (ii) For any process Pr_k, suppose the token is on an arc α , with corresponding input assertion ${}_iI^k_\alpha$ true. Then for all flowchart nodes not in Pr_k, execution of the node must preserve the truth of $(at\,\alpha \wedge {}_iI^k_\alpha)$. These conditions ensure that, with respect to an annotation, the entire program is consistent. ([LAM1] calls an annotation of a flowchart a 'generalized interpretation').

The consistency of the interpretation of an entire program is exactly similar to the interference-freedom of the proof outline of a program in [OG].

For two assertions P,Q about a program, [LAM1] gives two axioms for the relation $\rightsquigarrow$ (leads to).

Axiom L1: For a generalized interpretation of a program, if

(a) by assuming the invariance of $\neg Q$ it can be proved that,

(i) $P \supset (at\,\alpha \supset {}_iI^k_\alpha)$ for all arcs α, in all processes Pr_k; i.e. if the token of any process is on an (arbitrary) arc and P is true, then the corresponding input assertion is true.

(ii) the generalized interpretation for the entire program is consistent.

(b) $\{\alpha \mid {}_oI^k_\alpha = \text{false}\}$ is an inevitable set.

(An inevitable set for a program is a set of arcs,

such that for some process Pr_k, every closed path in process Pr_k contains an arc in the set, and all exit arcs of process Pr_k belong to the set).

Then $P \leadsto Q$.

This axiom describes the method of ensuring, for a given program and pair of assertions P, Q, that $P \leadsto Q$.

The idea is to assume the invariance of $\neg Q$, and show that starting in any state with P true, always ends up in a contradiction. Under the invariance of $\neg Q$, some arcs may be annotated with the output assertion 'false', because the token can never move to such an arc without violating $\neg Q$. Further, if forsome process Pr_k, every loop and exit arc is cut by an arc with output assertion 'false', it follows that every execution sequence starting in a state with P true, ends up with the token of process Pr_k moving to an arc with output assertion 'false'. Hence, the conclusion, $P \leadsto Q$ (i.e., the falsity of $P \wedge \Box \neg Q$ is equivalent to $P \supset \nabla Q$).

Axiom L2: (a) The relation $P \leadsto Q$ is transitively closed.

(b) If S_P is a finite set of assertions, and $P \leadsto Q$ for each $P \in S_P$, then $(\bigvee S_P) \leadsto Q$.

The following derived theorems are also of use,

Theorem: If $\Vdash (A \supset B)$ ($A \supset B$ is invariant), then $A \leadsto B$.

Theorem: If $\Vdash \neg C$ ($\neg C$ is invariant) implies $A \leadsto B$, then $A \leadsto B \vee C$.

Theorem: If $\Vdash C$ (C is invariant) and C is monotone (execution of every flowchart node preserves C) implies $A \leadsto B$ then $A \wedge C \leadsto B \wedge C$.

Theorem: Let S_P be a finite well founded set of assertions, (with an irreflexive, partial ordering '$<$'), and Q be any assertion.

If $P \leadsto (\vee P_<) \vee Q$, for all $P \in S_P$, then $(\vee S_P) \leadsto Q$.

(Note: $P_< \triangleq \{X \mid X \in S_P \wedge (X < P)\}$).

The major difference between [LAM1] and [OL], [MP] is that in [LAM1] the entire program must be examined to derive $P \leadsto Q$, whereas in the other methods, rules are given to derive $P \leadsto Q$ for a program fragment. Of course, in order to derive $P \leadsto Q$ for two arbitrary assertions P,Q, the actions of the entire program are relevant. However, in [OL],[MP] this global interaction of the entire program is captured by means of safety properties (invariants). These safety properties must be derived first, but their derivation is done completely independently of the liveness properties. In [OL], [MP], once the required safety properties are shown indeed to hold, $P \leadsto Q$ is derived trivially by applying one of the given rules.

A related point is that, in both [OL], [MP], the assertion P must be the conjunction of a location predicate and (possibly) some other assertion (i.e. $P \equiv \text{at } \ell \wedge \emptyset$). The location predicate identifies a program fragment, to which a liveness rule is applied. Such a constraint on P is obviated in [LAM1] by condition (a)(i) of axiom L1.

The methods of [OL] and [MP] are similar in many respects. Both use Temporal Logic to state and reason about liveness properties. Both give rules for deriving the most elementary liveness properties for a fragment of a given program.

The more complex liveness properties are derived from the more elementary ones by the standard techniques of induction and enumeration (case analysis) and by making use of the transitivity of $\leadsto$.

The rules are given for program fragments, which in terms of [OL] are atomic or compound statements, whereas in [MP] a program fragment is a set of control locations and all the associated transitions. The [OL] method is used with a programming language in which all constructs are single-entry, single-exit. There is no such restriction in [MP]. However, an arbitrary program can always be converted to one using only single-entry, single-exit constructs, so this difference is not at all significant. There is one major difference between the two methods: [OL] is based on the use of invariant assertions, whereas [MP] uses intermittent assertions.

2.4 Invariant Assertions-Intermittent Assertions

An invariant assertion is an assertion associated with a control location, such that the assertion is true every time control reaches that control location. An intermittent assertion need only be true sometime when control reaches the associated control location. An intermittent assertion is guaranteed to become true atleast once. An invariant assertion may never become true-that is, control may never reach its associated control location. This difference between [OL] and [MP] is illustrated by the following -

(Assume only single-entry, single-exit constructs are used).

Suppose, for a compound statement S, it is required to know the conditions under which it is guaranteed that control reaches after S with some assrtion Q true, i.e. the conditions required to derive ∇(after S $\wedge$ Q).

In [OL] the required conditions are

$$\frac{\{P\}\ S\ \{Q\},\ \Box(\text{in } S \supset P),\ \ \text{in } S \rightsquigarrow \text{after } S,\ \text{in } S}{\nabla(\text{after } S \wedge Q)}$$

Thus, an invariant assertion $\{P\}$ S $\{Q\}$ has to be established. The program control flow in S $\rightsquigarrow$ after S, is treated separately from assertions P,Q. The invariant $\Box$(in S $\supset$ P) is also required.

In [MP] the required conditions are

$$\frac{P \wedge \Box \text{in } S \ \supset \Box P,\ \text{in } S \wedge P \ \rightsquigarrow\ \text{after } S \wedge Q,\ \text{in } S \wedge P}{\nabla(\text{after } S \wedge Q)}$$

That is, P must be shown to be prese~~n~~r~~t~~ved as long as control is in S, the liveness must be shown between complex assertions (not simple location predicates), and initially P $\wedge$ in S must be shown to hold in contrast to in S alone as required in [OL].

In a method using invariant assertions, the two properties of partial correctness (i.e. $\{P\}$ S $\{Q\}$) and termination (i.e. in S $\rightsquigarrow$ after S) are regarded as distinct properties. These two properties are combined into a single liveness property (i.e. in S $\wedge$ P $\rightsquigarrow$ after S $\wedge$ Q) in a method which uses intermittent assertions.

Manna and Waldinger [MW], point out that any proof derived using invariant assertions may be converted (almost trivially) to one using intermittent assertions, but the converse is not true. A method using intermittent assertions is then more general than one using invariant assertions.

However, the naturalness and power of an intermittent assertion method is achieved at the cost of intertwining the properties of the abstract objects manipulated by the program and the control flow of the program itself. This issue was raised by Gries [GRI 2], who maintains that such an intertwining of properties of abstract objects and program control flow, contradicts the desire for a separation of concerns and, hence, concludes that the invariant assertion method is superior (for sequential programs).

In sum, an intermittent assertion method is more natural - in the sense of being akin to informal reasoning. But, a proof derived by this method, may miss out the additional insight gained, by using an invariant assertion method.

CHAPTER 3

EXAMPLE PROOFS

3.1 On-the-Fly Garbage Collector.

This is a two-process program to collect garbage in a list processing system - the program development is described in [DIJ 2] and a correctness proof using the Owicki-Gries method, is presented in [GRI 1]. The two processes are called the 'mutator' and the 'collector', and their only indivisible action need be the memory reference.

It is not a good program. The fine degree of interleaving makes the correctness proof very difficult to understand. Minor changes, seemingly of no consequence (eg. interchanging the two actions in procedures 'addleft' or 'addright'), give rise to subtle errors. Yet, because of these very reasons, it is of interest to examine this program.

One reason for difficulty in understanding this program is that, use is made of the properties of a directed graph with nodes of three colours, whose edges change over time. These properties are not at all well known and neither are they intuitively apparent - possibly if these properties are proved separately the correctness argument would be simplified.

The data structure used in a conventional implementation of LISP is a directed graph in which each node has at most two outgoing edges (either of which may be missing) - an outgoing left edge and an outgoing right edge. At any moment all nodes of the graph must be reachable, via a directed path, from a fixed root,

which has a fixed, known place in memory. The storage allocated for each node is constant in size and can accomodate two pointers, one for each outgoing edge. A special value NIL denotes a missing edge. The directed graph may have cycles.

For any reachable node an outgoing edge may be deleted, changed or added. Deletion and change may turn formerly reachable nodes into unreachable nodes which can no longer be used by the program (henceforth called the mutator). These unreachable nodes are called garbage. Nodes not being used by the mutator are stored in a Free List, maintained as a singly linked list. The mutator may take a single node from the free list, at a time. It does this by deleting the first node from the free list, and adding an edge to this node from a reachable node.

If the free list becomes empty, computation halts and a procedure called 'garbage collection' is invoked. Beginning with the root all reachable and free list nodes are marked. Upon completion of this marking phase, all unmarked nodes are known to be garbage and are appended to the free list. Computation is then resumed.

To avoid the disadvantage of the unpredictable garbage collection interludes, a second processor, the 'collector' is used concurrently with the mutator, to collect garbage on a more continuous basis.

Three constraints were placed on the desired program

(i) Interference between collector and mutator should be minimum.

(ii) The overhead on mutator activity should be as small as possible.

(iii) The ongoing activity of the mutator should not impair the collector's ability to identify garbage as soon as possible.

The program designed in [DIJ 2] is as follows. The collector has two phases - marking reachable nodes and collecting unmarked, unreachable nodes. Three colours are used for marking: white represents unmarked, black marked and gray an inbetween colour needed for mutator-collector co-operation.

The graph nodes are represented by an array m [0..N] for the nodes. NIL is represented by 0 and thus the mutator itself may never reference node m [0]. Each node has three sub-fields of interest, m[i]. Colour, the current colour of the node, m [i]. Left, the node's left son and m[i]. right, the node's right son.

Two nodes m [ROOT] and m [FREE] are in fixed, constant places in the array m. m[ROOT] is the single root of the mutators' graph, while m[FREE] is used to indicate where the free list begins. An extra integer variable ENDFREE is used to point to the last node in the free list. m[FREE] is not a free list node, while m [ENDFREE] is one. Nodes are coloured by one of the three indivisible actions,

Whiten (i) m[i]. colour: = white

Blacken (i) m[i]. colour: = black

atleast-gray (i)   if m[i]. colour = white then m [i].
colour: = Gray

(a black node is not affected by this action)

The mutator uses two procedures to add edges from one node to another -

```
Proc Addleft  (k, j);
{Add a left outgoing edge from node k to node j}
begin m[k]. left: = j ; atleast-gray (j) end;
Proc Addright (k,j);
{Add a right outgoing edge  from node k to node j}
begin m[k]. right: = j; atleast-gray (j) end;
```

That is, after adding an edge, but before attempting any other action, the mutator grays the destination node,of the added edge.

The mutator is in a never ending loop, repeatedly choosing one of the actions available to it (in a nondeterministic fashion). The mutator goes into a busy-wait loop, if it requires a free list node and the free list has only one node. This is the only synchronisation between mutator and collector.

```
    Program Garbage-Collector;
    Array: m [O..N] of node;
    ROOT, FREE, ENDFREE:O..N; i,j,kf: integer;
S:  Initialise ROOT, FREE; m[O]. left: = O, m[O]. right: = O;
    i: = N+1;
    Put all nodes (except O,ROOT,FREE) in freelist,colouring
    them white, with ENDFREE pointing to the last node;
sc: Cobegin
    Mutator || Collector
    Coend
```

Process Collector

c_0: atleast-gray (ROOT)

c_1: atleast-gray (FREE)

c_2: atleast-gray (0)

Mark Phase

c_3: i: = 0

c_4: If i =N+1 go to c_{12}

c_5: If m[i]. Colour=gray goto c_8

c_6: i: = i + 1

c_7: go to c_4

c_8: atleast gray (m[i]. left)

c_9: atleast gray (m[i]. right)

c_{10}: ⟨blacken (i), i:=0⟩

c_{11}: go to c_4

c_{12}: i: = 0

Collect Phase

c_{13}: If i=N+1 go to c_0

c_{14}: If m[i]. colour=Black goto c_{21}

c_{15}: m[i]. left: = 0

c_{16}: m[i]. right: = 0

c_{17}: m[ENDFREE]. left:=i

c_{18}: ENDFREE:=i

c_{19}: i: = i+1

c_{20}: go to c_{13}

c_{21}: Whiten (i)

{k, j are indices of nodes reachable from ROOT, $k \neq 0$, $j \neq 0$}

Process Mutator

{Nondeterministic Location}

m_0: go to $m_1, m_3, m_5, m_7, m_9, m_{15}$

{Delete left son}

m_1: m[k]. left:=0

m_2: go to m_0

{Delete right son}

m_3: m[k].right:= 0

m_4: go to m_0

{Add left son}

m_5: Add left (k,j)

m_6: go to m_0

{Add right-son}

m_7: Add right (k,j)

m_8: go to m_0

{Add left son from free list}

m_9: f:= m[FREE]. left

m_{10}: add left (k,f)

m_{11}: If f=ENDFREE goto m_{11}

m_{12}: addleft (FREE,m[f] .left)

m_{13}: m[f]. left: = 0

m_{14}: go to m_0

{Add right son from free list}

m_{15}: f:=m[FREE]. left

m_{16}: add right (k,f)

c_{22}: i: = i+1

c_{23}: go to c_{13}

m_{17}: If f=ENDFREE goto m_{17}

m_{18}: addleft (FREE, m[f].left)

m_{19}: m[f]. left: = 0

m_{20}: go to m_o

Notation used in proof-

Reach (x) $\equiv$ (there is a directed path from ROOT to x, or from FREE to x) $\vee$ (x = 0)

UnReach (x) $\equiv \neg$ Reach (x) (note that Unreach (x) $\supset x \neq 0 \wedge x \neq$ ROOT $\wedge\, x \neq$ FREE)

Black (x) $\equiv$ m[x]. colour = Black, similarly for White (x), Gray (x).

Lx = m[x]. left , Rx = m[x]. right

#Black = The number of black nodes in array m[0..N].

at Mark $\equiv$ at $\cancel{c_3}$ $c_4 \ldots c_{12} \equiv$ ~~at c_3~~ $\vee$ at $c_4 \vee$ at $c_5 \vee \ldots \vee$ at c_{12}

at Collect $\equiv$ at $c_{13} \ldots c_{23} \equiv$ at $c_{13} \vee$ at $c_{14} \vee$ at $c_{15} \vee \ldots \vee$ at c_{23}

Gray-reachable (x) $\equiv$ there is a directed path $(k_1, k_2, \ldots, k_p, x)$ where k_1 is gray and $k_2, \ldots, k_p$ are white.

Node k_1 is called a "Gray-Source" of node x.

A node y is called an ancestor of node x if Anc (x,y) is true.

Anc (x,y) $\equiv$ (y=x) $\vee$ Anc (x, Ly) $\vee$ Anc (x, Ry).

i.e. Anc (x,y) is true if there is a directed path from node y to x.

The subgraph consisting of nodes $\{y \mid \text{Anc}(x,y)\}$ need not be a tree-it may have one or more cycles.

Property: Unreach $(x) \supset \forall y\, [\text{Anc}\ (x,y) \supset \text{Unreach}\ (y)]$

i.e. all ancestors of an unreachable node are also unreachable. Obviously, if any ancestor of x were reachable, x would also be reachable.

The safety properties used are -

Q1 : at sc $\supset \Box$ at collector $\wedge$ $\Box$ at Mutator,

The collector and mutator are cyclic processes, so that once control reaches them, it stays in them forever.

Q2 : Unreach (x) $\wedge$ $\Box \neg$(at $c_{18} \wedge$ L ENDFREE $=x=i) \supset \Box$ Unreach (x), for all x.

An unreachable node is collected only by the collector action at c_{17} - otherwise, if this action never occurs, it stays forever unreachable.

Q3 : at s $\supset \Box$ (at $c_o \supset \forall_x$ ($\neg$Black (x)))
$\wedge \Box$ (at $c_{12} \supset \forall_x$ ($\neg$Gray(x) $\wedge$ (white(x) $\supset$ Unreach(x))))

This is the fundamental safety property. At the start of Mark Phase (c_o) all nodes are non-black. At the start of collect Phase (c_{12}) all nodes are non-gray and all white nodes are unreachable.

Q4 : $\forall y$ (Anc $(x,y) \supset \neg$Gray(y)) $\wedge$ $\Box$Unreach (x) $\supset$
$\Box \forall_y$(Anc $(x,y) \supset \neg$ Gray (y)), for all x.

This is of the form $P \wedge \Box R \supset \Box P$.

If all ancestors of node x are nongray, they remain forever non-gray, provided x is forever unreachable.

Q5 : $\forall y\,(\mathrm{Anc}\ (x,y) \supset \mathrm{White}\ (y)) \wedge \Box \mathrm{Unreach}\ (x) \supset$
$\Box \forall y\,(\mathrm{Anc}\ (x,y) \supset \mathrm{White}\ (y))$, for all x.

Again of the form $P \wedge \Box R \supset \Box P$.

If a node x is forever unreachable and all its ancestors are presently white, then they will all remain forver white.

This is actually a logical consequence of Q4 and "white nodes can only turn gray".

Two liveness lemmas are used -

L1 : $\cancel{\models \text{at } c_{13}..c_{12} \supset \nabla(\text{at } c_{13} \wedge i=0)}$ $\models \text{at } c_3..c_{11} \supset \nabla \text{at } c_{12}$

i.e. the Mark phase always terminates.

L2 : $\models \text{at } c_{13}..c_{23} \supset \nabla \ \text{at } c_o$

i.e. the Collect phase always terminates.

3.1.1 Proof of Safety Properties

Note that, to derive $P \wedge \Box R \supset \Box P$, it must be shown that $\{P \wedge R\}$ Any individual atomic action $\{P \vee \neg R\}$,

i.e. at $1 \wedge c_\alpha \wedge P\,(\bar{\pi};\ \bar{y}) \wedge R(\bar{\pi};\ \bar{y}) \supset \big[P(r_\alpha(\bar{\pi});\ f_\alpha(\bar{y}))$
$\vee \neg\ R(r_\alpha(\bar{\pi});\ f_\alpha(\bar{y}))\big]$

holds for every transition α,

$$\bigcirc \xrightarrow{\text{at } 1 \wedge c_\alpha \rightarrow [(\bar{\pi};\ \bar{y}) := (r_\alpha(\bar{\pi});f_\alpha(\bar{y}))]} \bigcirc \ .$$

Q2: Unreach (x) $\wedge \Box \neg(\text{at } c_{18} \wedge \text{L ENDFREE} = x = i) \supset \Box$ Unreach (x).

The mutator can only add edges to reachable nodes, hence it cannot affect this assertion.

The only action in which the collector adds an edge is at c_{17}, and after this action (at $c_{18} \wedge$ LENDFREE $= x = i$) is true, so that, it is indeed true that

$$\{\text{Unreach}(x) \wedge \neg(\text{at } c_{18} \wedge \text{LENDFREE}=x=i)\} \text{ Any atomic action } \left\{ \begin{array}{c} \text{Unreach}(x) \\ \vee \\ (\text{at } c_{18} \wedge \text{LENDFREE} \\ = x = i) \end{array} \right\}$$

Q3: $\text{at } s \supset \Box(\text{at } c_0 \supset \forall x\ (\neg \text{Black}(x)))$

$\wedge \Box(\text{at } c_{12} \supset \forall x\ (\neg\text{Gray}(x) \wedge (\text{White}(x) \supset \text{Unreach}(x))))$.

This is the fundamental safety property of the garbage collector. Three other invariants are required to prove it.

(R3) $\text{at } s \supset \Box\ (\text{at } c_4 .. c_{12} \supset \forall x(\text{White}(x) \wedge \text{Reach}(x) \supset \text{Gray-reachable}(x)))$,

During the mark phase every reachable white node is gray-reachable- i.e. there is a directed path from a gray node, along white nodes, to every reachable white node.

(G3) $\text{at } s \supset \Box(\text{at } c_4 .. c_{12} \supset [\exists \text{gray node in } m[0..i-1] \supset \exists \text{gray node in } m[i..N]])$

During a scan of array m, for the mark phase, if there is a gray node in the already scanned nodes $m[1..i-1]$, there must be a gray node among the nodes yet to be scanned, $m[i..N]$.

(B3) This assertion has been strengthened, so that it can also be shown that any node added to the Free List is indeed unreachable (this is a desired collector safety property).

$\text{at } s \supset \Box(\text{at } c_{13} ... c_{23}, c_0 \supset$

$\forall x\ [x \in \{0..i-1\} \supset \neg \text{Black}(x)]$

$\wedge \forall x\ [x \in \{i..N\} \supset \neg\text{Gray}(x) \vee (x=i \wedge \text{at } c_{19}, c_{22})]$

$\wedge \forall x\ [x \in \{i..N\} \wedge \text{White}(x) \supset \text{Unreach}(x) \vee (x=i \wedge \text{at } c_{18}, c_{19}, c_{22})]$

)

That is,

In the collect phase, there are no black nodes in the already scanned nodes $[1..i-1]$, and there are no gray nodes in $[i..N]$, and all white nodes in $[i..N]$ are unreachable.

Q3 is easily seen to be true from the three assertions (R3), (G3), (B3) as follows:

From (at $c_0 \supset i=N+1$) and (B3), $\Box(\text{at } c_0 \supset \forall x[x \in \{0..N\} \supset \neg \text{Black}(x)])$ (1)

From (at $c_{12} \supset i=N+1$) and (G3), $\Box(\text{at } c_{12} \supset \neg\exists \text{gray node in } m[0..N])$ (2)

because $\exists$ Gray node in $m[N+1..N]$ ~~$\vee(x=1)$~~ is obviously false. $\neg\exists$ gray node in m $[0..N] \supset \forall x \; (\neg \text{Gray-Reachable}(x))$, i.e. if there are no gray nodes at all, no node can be gray-reachable.

From this, and (R3),

$$\Box(\text{at } c_{12} \supset \forall x[\neg \text{White}(x) \vee \text{Unreach}(x)]) \quad (3)$$

Q3 is the conjunction of (1), (2) and (3).

The three assertions (R3), (B3), (G3) must be considered separately.

(R3) is true when control enters Mark phase, because ROOT, FREE have been grayed and all nodes are nonblack. Subsequently, while control is in Mark, (R3) can be falsified only by blackening a gray node which is a unique gray-source for some reachable, white node. However, when node i is blackened at c_{10}, both its sons have been previously grayed at c_8, c_9. Consequenrly, at the instant when the action at c_{10} occurs either both its sons are non-white, or the mutator has just added a white son to node i but not yet grayed it. In the latter case, the new white son

must have been gray-reachable before the mutator action (note that mutator only affects reachable nodes), so it has another gray-source besides node i.* In any case (R3) is preserved.

(G3) is trivially true when control enters Mark phase, because i=O ~~at c_3 occurs in the consequent.~~ Subsequently, it may be falsified by any action which

(i) Grays a node in [1..i-1] at a time when there are no other gray nodes in [1..i-1]

(ii) Blackens a gray node in [i..N] or decreases i

(iii) Increases i, that is the action at c_6.

If the mutator grays a white node in [1..i-1], at a time when there are no other gray nodes in [1..i-1], then the gray-source of this node must be present in [i..N].

The action which blackens a node in [i..N] sets i to 0, thereby making (G3) trivially true.

The action at c_6, which increments i, maintains (G3) because

$$\text{at } c_5 \wedge (G3) \wedge \neg \text{Gray (i)} \supset \text{at } c_5 \wedge (G3)^{i}_{i+1}$$

That is, when control first reaches c_6, (G3) holds with i replaced by i+1. This can be seen from,

$$\text{at } c_5 \wedge [\neg\exists \text{Gray node in } [1..i-1] \vee \exists \text{Gray node in } [i..N]] \wedge \neg \text{ Gray (i)}$$

$$\equiv \text{at } c_5 \wedge [\neg\exists \text{Gray node in } [1..i-1] \wedge \neg \text{Gray (i)} \vee \exists \text{Gray node in } [i..N] \wedge \neg\text{Gray (i)}]$$

$$\equiv \text{at } c_5 \wedge (G3)^{i}_{i+1}.$$

(*Note: The other gray source of the new white son cannot be node i, because node i's second son is non-white).

Hence, if the transition at c_5, leading to c_6 occurs, $(G3)^{i}_{i+1}$ must be true immediately before and after the transition. Subsequently, after the action at c_6, (G3) again holds by rule of assignment i.e. $\{(G3)^{i}_{i+1}\}$ i: = i+1 $\{(G3)\}$.

(B3) is true when control enters $c_{13}..,c_{23},c_o$ because i is set to 0 by the action at c_{12}, and by (R3), (G3) it follows that all nodes are non-gray and all white nodes are unreachable, and control first reaches c_{13} through c_{12}. It must be shown that (B3) holds subsequently. (B3) actually consists of three subassertions,

(B3b) - at $c_{13}...c_{23},c_o \supset \forall x\,[\,x \in \{0..i-1\} \supset \neg\,\text{Black}\,(x)\,]$

The mutator cannot affect (B3b).

This is maintained subsequently because the only collector actions to affect it (in collect phase) are the increments of i at c_{19}, c_{22}. The action at c_{19} is always preceded by a check at c_{14} that node i is nonblack, and the action at c_{22} is always preceded by the whitening of node i.

(B3w) - at $c_{13}..c_{23},c_o \supset \forall x\,[\,x \in \{i..N\} \wedge \text{White}(x) \supset (\text{Unreach}(x) \vee (x=i \wedge \text{at } c_{18},c_{19},c_{22}))\,]$

The mutator cannot affect (B3w). The collector actions affecting it (in collect phase) are those that change i, and those that falsify $(\text{unreach }(x) \vee (x=i \wedge \text{at } c_{18},c_{19},c_{22}))$. The action at c_{17} does falsify unreach (i) but makes 'at c_{18}' true. The actions at c_{19}, c_{22} increment i and falsify 'at c_{18},c_{19},c_{22}', but incrementing i strengthens the antecedent of (B3w) so that the truth of (B3w) is maintained.

$$(B3g)\ \text{at } c_{13}..c_{23}, c_o \supset \forall x\ [\ x \in \{i..N\} \supset (\neg Gray(x) \vee (x{=}i \wedge \text{at } c_{19}, c_{22}))\ ]$$

The only collector actions to affect (B3g) are the ones at c_{19}, c_{22}, which both increment i and falsify at c_{19}, c_{22}. Again, an increment of i only strengthens the antecedent of (B3g), so that the assertion's truth is preserved.

The only way the mutator can falsify (B3g) is by graying a white, reachable node. However, from (B3w) it follows that a white reachable node x, can exist in i..N only for ($x{=}i \wedge$ 'at c_{18}, c_{19}, c_{22}'). The case $(x{=}i \wedge \text{at } c_{19}, c_{22})$ occurs in the consequent of (B3g), so in this case (B3g) is not affected by the graying of the white node i. The case ($x{=}i \wedge$ at c_{18}) also cannot lead to violation of (B3g) because the mutator cannot access a node to the left of ENDFREE, and at c_{18} the only white reachable node in $\{i..N\}$, the node i, is the left son of node ENDFREE.

$$Q4: \forall y\ (Anc(x,y) \supset \neg Gray(y)) \wedge \Box Unreach(x) \supset \Box \forall y\ (Anc(x,y) \supset \neg Gray(y))$$

From the definition of the relation Ancestor, it follows that

-All ancestors of x must be forever unreachable, if x is forever unreachable.

No mutator action can affect an unreachable node, so the mutator cannot gray any ancestor of x.

The collector only grays nodes by the actions at c_8, c_9 in the Mark Phase, but this is only done after checking at c_5 that the parent of these two nodes is gray. Since all the ancestors of x are nongray, there cannot be a node among the ancestors of x

whose parent is gray. Hence, the collector under the given conditions, can never gray any ancestor of x.

Q5: $\forall y\,[\mathrm{Anc}(x,y) \supset \mathrm{White}(y)] \wedge \Box\mathrm{Unreach}(x) \supset \Box\forall y\,[\mathrm{Anc}(x,y) \supset \mathrm{White}(y)]$

Q5 may be derived by considering all actions that may falsify it, as was done for Q4.

However, its truth follows from Q4 and the assertion "A white node can only turn gray".

From Q4 and the antecedent of Q5 follows that the ancestors of x are forever nongray. Since, initially the ancestors of x are all white, and they never turn gray, they must be forever white.

$$\forall y\,[\mathrm{Anc}(x,y) \supset \mathrm{White}(y)] \wedge \Box\,\mathrm{Unreach}(x)$$
$$\supset \forall y\,[\mathrm{Anc}(x,y) \supset \neg\mathrm{Gray}(y)] \wedge \Box\mathrm{Unreach}(x)$$
$$\supset \Box\forall y\,[\mathrm{Anc}(x,y) \supset \neg\mathrm{Gray}(y)] \text{ by Q4.}$$

From,

$$\forall y\,[\mathrm{Anc}(x,y) \supset \mathrm{white}(y)] \wedge \Box\mathrm{Unreach}(x) \wedge \Box\forall y\,[\mathrm{Anc}(x,y) \supset \neg\mathrm{Gray}(y)]$$

to conclude that $\Box\forall y\,[\mathrm{Anc}(x,y) \supset \mathrm{white}\,(y)]$, two assertions are required -

the first is the one mentioned above i.e.

$\mathrm{White}\,(x) \supset \neg\mathrm{Gray}\,(x)\,\Box\,\mathrm{White}\,(x)$, for all x.

the second is,

$\Box\,\mathrm{Unreach}(x) \supset \Box\forall y\,[\neg\,\mathrm{Anc}(x,y) \supset \Box\neg\,\mathrm{Anc}(x,y)]$, for all x.

The second assertion says that, if node x is forever unreachable, and at any instant a node y becomes a non-ancestor of x, then y remains forever a non-ancestor of x. That is, no node can turn into the ancestor of a forever unreachable node.

3.1.2 Proof of Liveness Properties

To prove the liveness lemmas, the following invariant on the value of variable i is used,

$$(\text{Qi})\ \text{at } S \supset \Box([\text{at } c_0 \ldots c_3, c_{12} \supset i=N+1] \wedge [\text{at } c_4, c_7, c_{13}, c_{20}, c_{23} \supset 0 \leqslant i \leqslant N+1]$$
$$\wedge [\text{at } c_5, c_6, c_8 \ldots c_{11}, c_{14} \ldots c_{19}, c_{21}, c_{22} \supset 0 \leqslant i \leqslant N]).$$

The following Induction Rule is used,

Induction allows $\models (\exists k{:}\emptyset(k)) \supset \nabla \emptyset'$, to be established from

$$\models \emptyset(0) \supset \emptyset' \quad \text{and} \quad \forall n{:}[\emptyset(n) \supset \nabla(\emptyset(N-1) \vee \emptyset')].$$

Lemma (L1)

$$\models \text{at } c_3 \ldots c_{11} \supset \nabla \text{ at } c_{12}$$

This is shown by SP rule applied to path $c_3 \longrightarrow [c_4 \ldots c_{11}] \longrightarrow c_{12}$
Induction must be used to show that control does leave $c_4 \ldots c_{11}$.

(1) $\models \text{at } c_3 \supset \nabla(\text{at } c_4 \wedge i=0)$ by ESC rule.

(2) $\models \text{at } c_4 \wedge i=N+1 \supset \nabla \text{ at } c_{12}$ by ESC rule.

Now induction is used to show

$$\text{at } c_4 \ldots c_{11} \supset \nabla(\text{at } c_4 \wedge i=N+1).$$

Let the 'bounding function' f be,

$$f(i, \#\text{Black}) \triangleq (N+1)[N - \#\text{Black}] + [(N+1) - i]$$

This is motivated by the fact that each iteration of Mark Phase increments i by 1 or blackens exactly one node (-the node i).

$$f = 0 \supset i = N+1 \wedge \#\text{Black} = N.$$

Basis (3) $\models$ at $c_4 \ldots c_{11} \wedge f=0 \supset$ at $c_4, c_7 \wedge i=N+1$ using (Qi).

(4) $\models$ at $c_4, c_7 \wedge i=N+1 \supset \nabla(\text{at } c_4 \wedge i=N+1)$ by ESC rule at c_7.

(5) $\models$ at $c_4 \wedge f=f_o \wedge i<N+1 \supset \nabla(\text{at } c_5 \wedge f=f_o \wedge i<N+1)$ by ESC rule.

(6) $\models$ at $c_5 \wedge f=f_o \wedge i<N+1 \supset \nabla([\text{at } c_6 \vee \text{at } c_8] \wedge f=f_o \wedge i<N+1)$ by ESC rule.

(7) $\models$ at $c_6 \wedge f=f_o \wedge i<N+1 \supset \nabla(\text{at } c_7 \wedge f=f_o-1 < f_o \wedge i \leq N+1)$ by ESC rule.

(8) $\models$ at $c_7 \wedge f=f_1 \wedge i<N+1 \supset \nabla(\text{at } c_4 \wedge f=f_1 \wedge i<N+1)$ by ESC rule.

(9) $\models$ at $c_8, c_9, c_{10} \wedge f=f_o \wedge i<N+1 \supset \nabla(\text{at } c_{11} \wedge f=f_o-(N+1-i)<f_o \wedge i=0)$ by SP rule to path $c_8 \rightarrow c_9 \rightarrow c_{10} \rightarrow c_{11}$.

(10) $\models$ at $c_{11} \wedge f=f_2 \wedge i<N+1 \supset \nabla(\text{at } c_4 \wedge f=f_2 \wedge i<N+1)$ by ESC rule.

(11) $\models$ at $c_6 \wedge f=f_o \wedge i<N+1 \supset \nabla(\text{at } c_4 \wedge f<f_o \wedge i<N+1) \vee \nabla(\text{at } c_4 \wedge i=N+1)$ by (7), (8) and (4).

(12) $\models$ at $c_8, c_9, c_{10} \wedge f=f_o \wedge i<N+1 \supset \nabla(\text{at } c_4 \wedge f<f_o \wedge i<N+1)$ by (9), (10).

(13) $\models$ at $c_6, c_8, c_9, c_{10} \wedge f=f_o \wedge i<N+1 \supset \nabla(\text{at } c_4 \wedge f<f_o \wedge i<N+1) \vee \nabla(\text{at } c_4 \wedge i=N+1)$ by (11), (12).

(14) $\models$ at $c_5, c_6, c_8, c_9, c_{10} \wedge f=f_o \wedge i<N+1 \supset \nabla(\text{at } c_4 \wedge f<f_o \wedge i<N+1) \vee \nabla(\text{at } c_4 \wedge i=N+1)$ by (6), (13).

(15) $\models$ at $c_4, c_5, c_6, c_8, c_9, c_{10} \wedge f=f_o \wedge i<N+1 \supset \nabla(\text{at } c_4 \wedge f<f_o \wedge i<N+1) \vee \nabla(\text{at } c_4 \quad i=N+1)$ by (5), (14).

(16) $\models$ at $c_4 \ldots c_6, c_8 \ldots c_{11} \wedge f=f_o \wedge i<N+1 \supset \nabla(\text{at } c_4 \wedge f<f_o \wedge i<N+1) \vee \nabla(\text{at } c_4 \wedge i=N+1)$ by (10), (15).

(17) $\models \text{at } c_4 \ldots c_{11} \wedge f=f_o \wedge i<N+1 \supset \nabla(\text{at } c_4 \wedge f<f_o \wedge i<N+1)$
$\vee \nabla(\text{at } c_4 \wedge i=N+1)$ by (8), (16).

(18) $\models (\text{at } c_4 \ldots c_{11} \wedge f=f_o \wedge i<N+1) \wedge (\text{at } c_4, c_7 \wedge i=N+1) \supset$
$\nabla(\text{at } c_4 \wedge f<f_o \wedge i<N+1) \vee \nabla(\text{at } c_4 \wedge i=N+1)$ by (4), (17).

This , using (Qi), gives the required Induction Step

(19) $\models \text{at } c_4 \ldots c_{11} \wedge f=f_o \supset \nabla(\text{at } c_4 \wedge f<f_o) \vee \nabla(\text{at } c_4 \wedge i=N+1)$ by (18), (Qi)

Using Induction

(20) $\models \text{at } c_4 \ldots c_{11} \supset \nabla(\text{at } c_4 \wedge i=N+1)$ by Basis i.e.((3),(4)) and (19)

(21) $\models \text{at } c_3 \ldots c_{11} \supset \nabla(\text{at } c_{12})$ by (1), (2), (20).

<u>Lemma (L2)</u> : $\models \text{at } c_{13} \ldots c_{23} \supset \nabla \text{ at } c_o$

Assume throughout j is some integer, $0 \leq j \leq N+1$.

(1) $\models \text{at } c_{13} \wedge i = N+1 \supset \nabla \text{ at } c_o$ by ESC rule.

Induction will be used to show

$(\text{at } c_{13} \ldots c_{23} \wedge i<j \leq N+1) \vee (\text{at } c_{13}, c_{20}, c_{23} \wedge i=j) \supset \nabla(\text{at } c_{13} \wedge i=j \leq N+1)$

Let the 'bounding function' f be defined by

$f(i) \triangleq j-i$

$f = 0 \supset i=j.$

Basis (2) $\models [(\text{at } c_{13} \ldots c_{23} \wedge i<j \leq N+1) \vee (\text{at } c_{13}, c_{20}, c_{23} \wedge i=j \leq N+1)] \wedge f=0$
$\supset (\text{at } c_{13}, c_{20}, c_{23} \wedge i=j \leq N+1)$, by $[f=0 \supset i=j]$

(3) $\models \text{at } c_{13}, c_{20}, c_{23} \wedge i=j \leq N+1 \supset \nabla(\text{at } c_{13} \wedge i=j \leq N+1)$
by ESC rule applied twice, to c_{20} and then to c_{23}.

(4) $\models$ at $c_{13}, c_{14} \wedge i < j \wedge f=f_o \supset \nabla([$at $c_{15} \vee$ at $c_{21}] \wedge i < j \wedge f=f_o)$

by ESC rule applied twice, to c_{13} and then to c_{14}.

(5) $\models$ at $c_{15} \cdot\cdot c_{19} \wedge i<j \wedge f=f_o \supset \nabla$(at $c_{20} \wedge i \leq j \wedge f=f_o - 1 < f_o$)

by SP rule applied to path $c_{15} \rightarrow c_{16} \rightarrow c_{17} \rightarrow c_{18} \rightarrow c_{19} \rightarrow c_{20}$.

(6) $\models$ at $c_{21}, c_{22} \wedge i \leq j \wedge f=f_o \supset \nabla$(at $c_{23} \wedge i \leq j \wedge f=f_o - 1 < f_o$)

by SP rule applied to path $c_{21} \rightarrow c_{22} \rightarrow c_{23}$.

(7) $\models$ at $c_{20}, c_{23} \wedge i \leq j \wedge f=f_1 \supset \nabla$(at $c_{13} \wedge i < j \wedge f = f_1$)

by ESC rule applied twice, to c_{20} and then to c_{23}.

(8) $\models$ at $c_{15} \cdot\cdot c_{19} \wedge i < j \wedge f=f_o \supset [\nabla$(at $c_{13} \wedge i<j \wedge f<f_o) \vee$ ∇ (at $c_{13} \wedge i=j$)] by (5), (7), (3).

(9) $\models$ at $c_{21}, c_{22} \wedge i < j \wedge f=f_o \supset [\nabla$(at $c_{13} \wedge i<j \wedge f < f_o) \vee$ ∇ (at $c_{13} \wedge i=j$)] by (6), (7), (3).

(10) $\models$ at $c_{15} \cdot\cdot c_{19}, c_{21}, c_{22} \wedge i < j \wedge f = f_o \supset [\nabla$(at $c_{13} \wedge i < j \wedge f < f_o) \vee$ ∇ (at $c_{13} \wedge i = j$)] by (8), (9).

(11) $\models$ at $c_{13} \cdot\cdot c_{19}, c_{21}, c_{22} \wedge i < j \wedge f=f_o \supset [\nabla$(at $c_{13} \wedge i<j \wedge f<f_o) \vee$ ∇(at $c_{13} \wedge i=j$)] by (4), (10).

(12) $\models$ at $c_{13} \cdot\cdot c_{23} \wedge i < j \wedge f=f_o \supset [\nabla$(at $c_{13} \wedge i < j \wedge f < f_o) \vee$ ∇ (at $c_{13} \wedge i=j$)] by (7), (11).

(13) $\models$ (at $c_{13} \cdot\cdot c_{23} \wedge i<j \wedge f=f_o) \vee$ (at $c_{13}, c_{20}, c_{23} \wedge i=j) \supset$ ∇ (at $c_{13} \wedge i<j \wedge f<f_o) \vee \nabla$(at $c_{13} \wedge i=j$) by (3), (12).

This is the required induction step. Hence by Induction

(14) $\models$ (at $c_{13} \cdot\cdot c_{23} \wedge i<j) \vee$ (at $c_{13}, c_{20}, c_{23} \wedge i=j) \supset \nabla$(at $c_{13} \wedge i=j$)

for all j, $0 \leq j \leq N+1$ by (2), (3), (13).

(15) $\models$ (at $c_{13} \cdot\cdot c_{23} \wedge i \leq N) \vee$ (at $c_{13}, c_{20}, c_{23} \wedge i=N+1) \supset$ ∇ (at $c_{13} \wedge i=N+1$) by (14), using $j=N+1$

(16) $\models$ at $c_{13}..c_{23} \supset \nabla$(at $c_{13} \wedge$ i=N+1) by (15), using (Qi)

(17) $\models$ at $c_{13}..c_{23} \supset \nabla$(at c_o) by (1), (16).

Main Liveness Proof

$\models$ Unreach (x) $\supset$ ∇ (ENDFREE = x), for all x.

i.e. every unreachable node is eventually added to the free list. By the definition of unreach, x $\neq$ ROOT $\wedge$ x $\neq$ FREE $\wedge$ x $\neq$ 0. Obviously, the distinguished nodes ROOT, FREE, 0 are never put in the free list.

In the following,

x $\in$ {0..N} - {0, ROOT, FREE}

(1) $\models$ Unreach(x) $\supset \Box$ UnReach(x) $\vee$ ∇ (at $c_{18} \wedge$ LENDFREE=x=i)

by safety prop (Q2).

(2) $\models$ at $c_{18} \wedge$ i=x $\supset \nabla$(at $c_{19} \wedge$ (ENDFREE=x)) by ESC rule.

(3) $\models \Box$ Unreach(x) $\supset \Box$ Unreach(x) $\wedge \Box$ in Collector by safety prop (Q1).

(4) $\models$ in collector $\equiv$ at $c_o..c_2 \vee$ at $c_3..c_{11} \vee$ at $c_{12} \vee$ at $c_{13}..c_{23}$.

(5) $\models$ at $c_{13}..c_{23} \supset \nabla$ at c_o by liveness Lemma (L2)

(6) $\models$ at $c_o..c_2 \supset \nabla$ at c_3 by SP rule to path $c_o \rightarrow c_1 \rightarrow c_2 \rightarrow c_3$

(7) $\models$ at $c_3..c_{11} \supset \nabla$ at c_{12} by liveness lemma (L1)

(8) $\models$ in collector $\supset \nabla$ at c_{12} by (4), (5), (6), (7)

(9) $\models \Box$ Unreach(x) $\supset \nabla$ at c_{12} $\wedge \Box$ Unreach (x) by (3), (8)

(10) $\models$ at $c_{12} \supset \forall$y ($\neg$Gray (y)) by safety prop (Q3)

(11) $\models$ at c_{12} $\wedge \forall$y ($\neg$Gray (y)) $\supset$ at $c_{12} \wedge \forall$y [(Anc(x,y) $\supset \neg$Gray(y)),]

for any node x.

(12) $\models$ at c_{12} $\wedge \Box$ Unreach(x) $\supset$ [at $c_{12} \wedge \Box$ Unreach(x) $\wedge$

$\forall$y (Anc(x,y) $\supset \neg$Gray(y))] by (10), (11)

(13) $\models \forall y\ (Anc(x,y) \supset \neg Gray(y)) \wedge \Box Unreach\ (x) \supset$
$\Box \forall y (Anc(x,y) \supset \neg Gray(y))$ the safety prop (Q4)

(14) $\models$ at $c_{12} \wedge \Box Unreach(x) \supset [\Box \forall y (Anc(x,y) \supset \neg Gray\ (y))$
$\wedge$ at $c_{12} \wedge \Box Unreach\ (x)]$ by (12), (13)

(15) $\models$ at $c_{12} \supset \nabla$(at $c_{13} \wedge i=0$) by ESC rule

(16) $\models$ at $c_{12} \supset \nabla$at c_o by lemma (L2), (15)

(17) $\models$ at $c_o \supset \forall y\ (\neg Black\ (y))$ by safety prop (Q3)

(18) $\models$ at $c_{12} \supset \nabla$ (at $c_o \wedge \forall y (\neg Black\ (y))$ by (16), (17)

(19) $\models$ [at $c_{12} \wedge \Box Unreach(x) \wedge \Box \forall y (Anc(x,y) \supset \neg(Gray\ (y))] \supset$
$[\nabla$ (at $c_o \wedge \forall y\ (\neg Black(y)) \wedge \Box Unreach(x) \wedge \Box \forall y (Anc(x,y) \supset$
$\neg Gray\ (y))]$ by (18)

(20) $\models \forall y\ (\neg Black\ (y)) \wedge \Box \forall y (Anc(x,y) \supset \neg Gray\ (y)) \supset$
$[\forall y (Anc\ (x,y) \supset White\ (y))]$, by $\forall y\ [white(y) \vee Gray\ (y) \vee Black(y)]$

(21) $\models$ [at $c_o \wedge \forall y\ (\neg Black\ (y)) \wedge \Box Unreach(x) \wedge \Box \forall y (Anc(x,y)$
$\supset \neg Gray(y))] \supset$ [at $c_o \wedge \Box Unreach(x) \wedge \forall y (Anc(x,y) \supset white(y))]$
by (20)

(22) $\models [\forall y\ (Anc(x,y) \supset white(y)) \wedge \Box Unreach(x)] \supset [\Box \forall y (Anc(x,y)$
$\supset white\ (y))]$, the safety Prop (Q3)

(23) $\models$ [(at c_o) $\wedge \Box Unreach(x) \wedge \forall y (Anc(x,y) \supset white(y))] \supset$
[at $c_o \wedge \Box Unreach(x) \wedge \Box \forall y (Anc(x,y) \supset White(y))]$ by (22)

(24) $\models \Box \forall y (Anc(x,y) \supset white(y)) \supset \Box White(x)$, by $\models Anc(x,x)$

(25) $\models$ [at $c_o \wedge \Box$ Unreach(x) $\wedge \Box \forall y\ (Anc(x,y) \supset white(y))] \supset$
[at $c_o \wedge \Box$ Unreach(x) $\wedge \Box$ white(x)] by (24)

(26) $\models$ at $c_o \supset \nabla$ (at $c_{13} \wedge i=0$) by (6), Lemma (L1), (15)

(27) $\models$ at $c_{13} \wedge i=0 < x \supset \nabla(\text{at } c_{13} \wedge i=x)$ by step (14) in proof of Lemma (L2), which was derived by Induction.

(28) $\models$ at $c_0 \wedge \Box \text{Unreach}(x) \wedge \Box \text{white}(x) \supset [\nabla (\text{at } c_{13} \wedge i=x) \wedge \Box \text{white}(x) \wedge \Box \text{unreach}(x)]$ by (26), (27)

(29) $\models$ at $c_{13} \wedge i=x \wedge \Box \text{white}(x) \supset \nabla(\text{at } c_{19} \wedge \text{ENDFREE}=x)$ by SP rule applied to path $c_{13} \rightarrow c_{14} \rightarrow c_{15} \rightarrow c_{16} \rightarrow c_{17} \rightarrow c_{18} \rightarrow c_{19}$ with $\chi \equiv$ white(x), $\emptyset \equiv (i=x)$.

*(30) $\models$ at $c_0 \wedge \Box \text{Unreach}(x) \wedge \Box \text{White}(x) \supset \nabla(\text{ENDFREE}=X)$ by (28), (29) and Temporal Logic.

(31) $\models$ at $c_0 \wedge \Box \text{Unreach} \wedge \forall y\ (\text{Anc}(x,y) \supset \text{white}(y)) \supset \nabla(\text{ENDFREE}=x)$ by (23), (25), (30)

*(32) $\models$ at $c_{12} \wedge \Box \text{Unreach}(x) \wedge \Box \forall y(\text{Anc}(x,y) \supset \neg \text{Gray}(y)) \supset \nabla(\text{ENDFREE}=x)$ by (19), (21), (31) and Temporal Logic.

(33) $\models$ at $c_{12} \wedge \Box \text{Unreach}(x) \supset \nabla(\text{ENDFREE}=x)$ by (14), (32).

*(34) $\models \Box\ \text{Unreach}(x) \supset \nabla(\text{ENDFREE}=x)$ by (9), (33) and Temporal Logic.

(35) $\models$ Unreach(x) $\supset \nabla(\text{ENDFREE}=x)$ by (1), (2), (34).

The main liveness proof seems long (and tedious). The argument used is quite simple.

Sometime after a node x becomes unreachable, control always reaches c_{12} in the collector. At c_{12}, all nodes are nongray and so all ancestors of x are also nongray. If x is assumed to be forever unreachable, this ensures, by (Q4) that all

---

*Note: in steps (30), (32), (34) the Temporal Logic Theorem

$$\nabla R \wedge \Box P \supset \nabla [R \wedge \Box P], \text{ is used.}$$

ancestors of x are from then onwards always nongray.

Control always goes from c_{12} to c_o, and at c_o all nodes are non-black. Thus at c_o, all ancestors of x are forever nongray and also presently non-black. Hence at c_o, all ancestors of x are white.

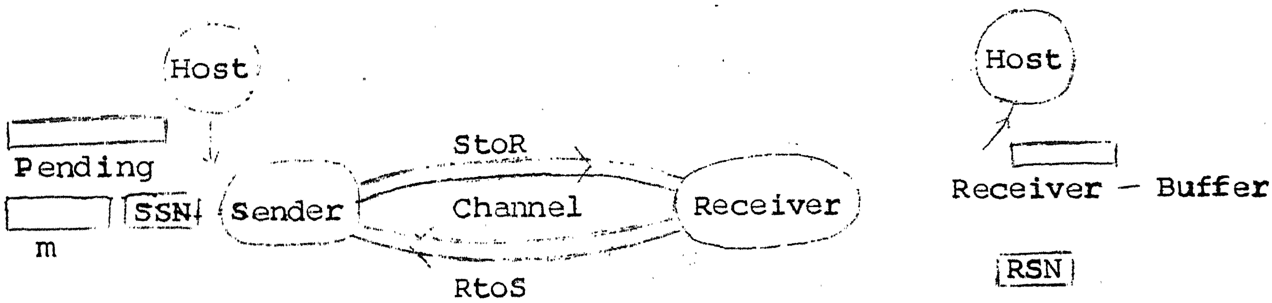
Again, since x is assumed forver unreachable, and all its ancestors are white, by (Q5), from this instant onwards all ancestors of x are forever white. Hence, from this instant onwards x is forever white.

Any node which is forever white must be collected and put into the free-list by the collect-phase (when i=x). Hence, x is eventually put into the free-list.

3.2 Alternating Bit Protocol

The protocol implementation and the safety proof follows [SUN]. This is a simplex protocol to send a message, m, from a sender station to a receiver station. Both sender and receiver maintain one-bit counters, called SSN, RSN respectively. The sender transmits a packet made of a sequence number field, which holds the present value of SSN, and a message field.

The receiver, on receipt of a packet whose sequence number field matches RSN, accepts the packet and flips RSN. It then delivers the message contained in the packet (to its host) and sends back an acknowledgement (ack). The ack is actually identical to the packet just received. The sender can proceed to the next message only after it gets the ack. Message packets, as well as acks, may be lost. Hence the sender keeps retransmitting message packets, until it gets the ack. On receipt of the expected ack, the sender flips its own one-bit counter, SSN.



Sent $\triangleq$ Sequence of all messages sent.

Received $\triangleq$ Sequence of all messages received.

The distributed protocol system has been modelled as a system with central shared memory, by assuming the medium

(or channel) to be a queue of packets.

The Sender to Receiver medium (StoR) and the Receiver to Sender medium (RtoS) are both queue-of-packets variables. Such a variable has the value Empty when the queue contains no packets. The operations on a queue-of-packets variable, q, are First(q), which returns the first element in the queue, Rest (q), which returns the queue without the first element and q @ P, which appends a packet p, to the rear end of the queue.

Pending and Receiver-Buffer are packet variables at the sender and receiver, respectively. They are used to buffer outgoing and incoming messages, respectively. A packet is a composite object with a message field and a sequence number field. A packet value can be constructed by Makepacket (message, sequence number). A packet variable has the value NIL when it holds no packet. The message field of a packet variable, p, is returned by the function Text (p), and the function Seq (p) returns the sequence number field.

In the Alternating Bit protocol, the sequence number is represented by a single bit, so it may only be 0 or 1.

$$\neg 0 = 1 \quad \text{and} \quad \neg 1 = 0.$$

In the following program, each process is described by a do-od construct. Two points should be made about this 'parallel do-od.

(i) If all the guards of a do-od construct are false, the process waits for some guard to become true
- i.e. the do-od does not terminate.

(ii) Each guarded command (i.e. guard and associated statement list) is an indivisible action.

That is, no action from another process may be interleaved between the evaluation of the guard (to True) and the associated statement list.

```
Program Alternating Bit Protocol;
RtoS, StoR: queue-of-packets modelling the two mediums;
Pending, Receiver-Buffer: Packet;
SSN, RSN: Sender and Receiver sequence number counters-
          one bit;
Sent, Received: queue of message, history variables,
recording the sequence of all messages sent and received;
M: Message, a buffer at the sender, in which Host places
message to be sent;
Host Ready: Boolean, set True by host after placing
          message in m;
(* Initialize Protocol*)
Pending, Receiver: = NIL, NIL,
RtoS, StoR       : = Empty, Empty,
Received, Sent   : = Empty, Empty,
RSN, SSN         : = 0, 0,
Host Ready       : = False

          Cobegin
Lose-Ack || Lose-Packet || SenderA || Sender-B || Receiver || Host-Proc

          Coend
```

```
Process Lose-Ack;

          do
          (* Lose first ack in queue RtoS*)
Lose-Ack:  ⟨RtoS ≠ empty → RtoS:= Rest (RtoS)⟩
          od


Process Lose Packet;

          do
          (* Lose first packet in queue StoR *)
Lose-Packet: ⟨StoR ≠ empty → StoR: = Rest (StoR)⟩
          od

Process Sender-A

          do
          (* Send a message, placed in m, by the host *)
Sender-A1: ⟨Pending = NIL ∧ Host-Ready →
          Pending: = Make Packet (m, SSN),
          StoR   : = StoR @ Make Packet (m, SSN),
          Sent   : = Sent @ m ⟩

          (*Get the expected ack from receiver*)
Sender-A2: ⟨RtoS ≠ empty ∧ Pending ≠ NIL ∧ Seq(First(RtoS))=SSN →
          Host-Ready: = False,
          Pending   : = NIL,
          SSN       : = ¬SSN,
          RtoS      : = Rest (RtoS)⟩
```

(* Get the ack for an old message- i.e. the last but one message sent-ignore it *)

Sender-A_3: $\langle$ RtoS $\neq$ empty $\wedge$ Pending $\neq$ NIL $\wedge$ Seq (First(RtoS)) = SSN $\rightarrow$

RtoS: = Rest (RtoS) $\rangle$

od

<u>Process Sender-B</u>

do

(*Retransmit a message whose ack has not yet been received*)

Sender-B: $\langle$ Pending $\neq$ NIL $\rightarrow$

StoR: = StoR @ Pending $\rangle$

od

<u>Process Receiver</u>

do

(* Receive the next message *)

Receiver_1: $\langle$ StoR $\neq$ empty $\wedge$ Receiver-Buffer = NIL $\wedge$ Seq(First (StoR)) = RSN $\longrightarrow$

Receiver-Buffer: = First (StoR)

RSN : = $\neg$ RSN,

StoR : = Rest (StoR) $\rangle$

(* Get an old message - i.e. one that has been already received and acknowledged - send a fresh ack for this message *)

Receiver_2: $\langle$ StoR $\neq$ empty $\wedge$ Receiver-Buffer=NIL $\wedge$ Seq(First(StoR)) = $\neg$ RSN $\longrightarrow$

RtoS: = RtoS @ First (StoR),

StoR: = Rest (StoR) $\rangle$

(* Deliver the message in Receiver Buffer-by appending it to Received- and acknowledge *)

Receiver$_3$: $\langle$ Receiver-Buffer $\neq$ NIL $\longrightarrow$

Received: = Received @ Text (Receiver-Buffer),

RtoS : = RtoS @ Receiver-Buffer,

Receiver-Buffer: = NIL $\rangle$

od

<u>Process Host Proc</u>

(* Host Proc describes the behaviour of the sender host-it is not a part of the Alternating Bit protocol system, and its actions are not considered in the correctness proof - its description may be taken to be a specification *)

do

(* The Host has no message to send *)

$\langle \neg$ Host-Ready $\longrightarrow$ Skip $\rangle$

(* The Host places a message in m, and sets Hostready to true *)

$\langle \neg$Host-Ready $\longrightarrow$ m: = message,

Host-Ready: = True $\rangle$

(* The Host waits for sender to complete transmission *)

$\langle$Host-Ready $\longrightarrow$ Skip $\rangle$

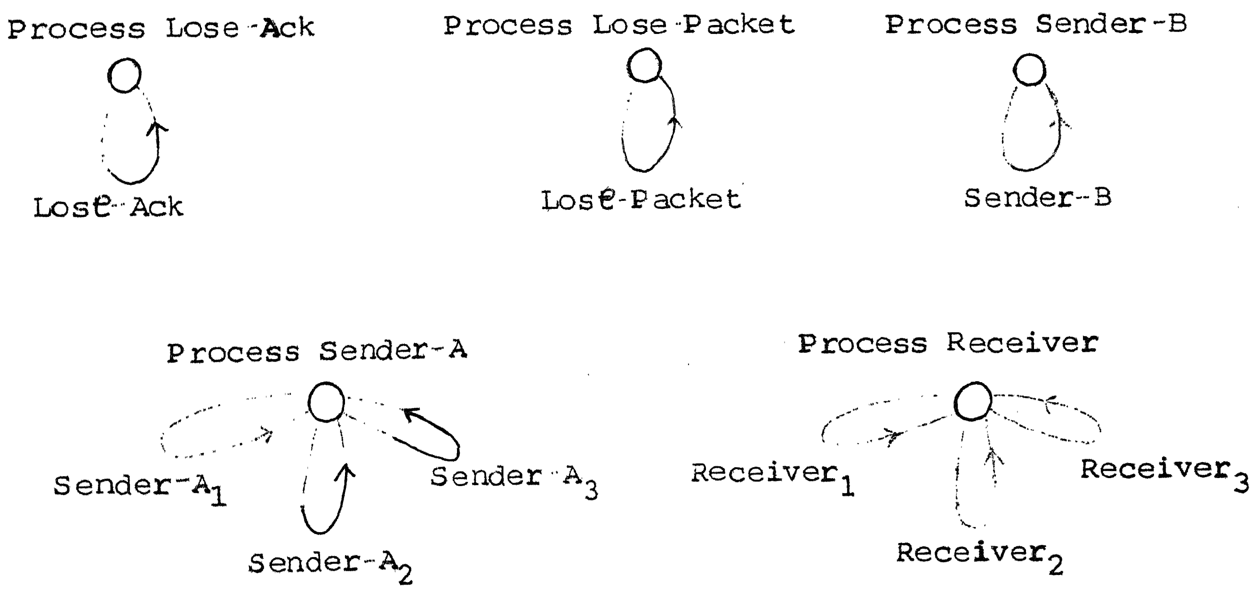
od

In this program, each process (Host Proc is not considered further) has only a single deterministic control location. Hence, after every transition, each process remains in its unique control location. If $\bar{l}_o$ is the vector of initial control locations, then

$$\text{at } \bar{l}_o \supset \Box \text{ at } \bar{l}_o.$$

This being the case, no location predicates are used in the derivations. The location predicates can easily be introduced, to make the derivations conform exactly to the ⌈MP⌉ rules.

Note that Sender has been split into two processes, so as to ensure that all locations are deterministic.



3.2.1 The Safety Properties

The Alternating Bit protocol system is shown to be always in one of four states. Liveness arguments are subsequently used,

to show that, the protocol system does indeed cycle through the four states. The four states are Relaxed, Sending, Send Complete, Acking.

The initial condition is described by

$$
\begin{aligned}
\text{Init} \triangleq\ & (\text{Pending-Receiver-Buffer} = \text{NIL} \\
& \wedge \text{Receiver} = \text{Sent} = \text{Empty} \\
& \wedge \text{RSN} = \text{SSN} = 0 \wedge \text{StoR} = \text{RtoS} = \text{Empty} \\
& \wedge \text{Host Ready} = \text{False})
\end{aligned}
$$

The required safety properties are

(I1) $\text{Init} \supset \Box (\text{Relaxed} \vee \text{Sending} \vee \text{Send Complete} \vee \text{Acking})$.

- the system is always in one of the four states. The states are such that, two of them cannot be true for the system together.

(I2) $\text{Relaxed} \supset \neg \text{Sending} \ \Box\ \text{Relaxed}$

(I3) $\text{Sending} \supset \neg \text{Send Complete} \ \Box\ \text{Sending}$

(I4) $\text{Send Complete} \supset \neg \text{Acking} \ \Box\ \text{Send Complete}$

(I5) $\text{Acking} \supset \neg \text{Relaxed} \ \Box\ \text{Acking}$

- the above four assertions use the dyadic $\Box$ operator and are of the form $I \supset R \ \Box\ I$, i.e. if I ever becomes true, then it remains true "as long as" R remains true. Thus, each assertion says that if the protocol system is in a particular state, then it continues to remain in that state, until it goes to the unique successor state.

The succession of states is given by

Relaxed $\rightarrow$ Sending $\rightarrow$ Send-Complete $\rightarrow$ Acking $\rightarrow$ Relaxed...

Deriving (I1)

Initial: (I1) is true initially because

$$\text{Init} \supset \text{Relaxed}.$$

Inductive: If (I1) is initially in one of the four states and (I2) to (I5) hold, then (I1) is obviously inductive. In order to derive (I2) to (I5), which are of the form $I \supset R \;\Box\; I$, it must be shown for all transitions α, that,

$$\text{at } l \wedge c_\alpha(\bar{y}) \wedge I(\bar{\pi};\ \bar{y}) \wedge R(\bar{\pi};\ \bar{y}) \supset \Big[I(r_\alpha(\bar{\pi}); f_\alpha(\bar{y})) \vee \neg R(r_\alpha(\bar{\pi});\ f_\alpha(\bar{y})) \Big]$$

(In terms of MP this would mean that I is R-invariant).

This verification condition is abbreviated

$$\{I \wedge R\}\ \alpha\ \{I \vee \neg R\}$$

The assertions defining the four states are

$$\begin{aligned} \text{Relaxed} \triangleq\ & \text{Pending} = \text{NIL} \wedge \text{SSN=RSN} \wedge \text{Receiver-Buffer=NIL} \\ & \wedge \text{Sent=Received} \wedge \text{StoR} \in \{(m', \neg \text{SSN})^*\} \\ & \wedge \text{RtoS} \in \{(m', \neg \text{SSN})^*\} \end{aligned}$$

$$\begin{aligned} \text{Sending} \triangleq\ & \text{Pending}=(m,\text{SSN}) \wedge \text{SSN=RSN} \wedge \text{Receiver-Buffer=NIL} \\ & \wedge \text{Sent=Received @ } m \wedge \text{StoR} \in \{(m', \neg \text{SSN})^+.(m,\text{SSN})^+ + (m,\text{SSN})^*\} \\ & \wedge \text{RtoS} \in \{(m', \neg \text{SSN})^*\} \end{aligned}$$

Send Complete $\triangleq$ Pending $=(m,SSN) \wedge SSN = \neg RSN \wedge$ Receiver-Buffer $=(m,SSN)$

$\wedge$ Sent = Received @ $m \wedge$ StoR $\in \{(m, SSN)^*\}$

$\wedge$ RtoS $\in \{(m', \neg SSN)^*\}$

Acking $\triangleq$ Pending = $(m,SSN) \wedge SSN = \neg RSN \wedge$ Receiver-Buffer=NIL

$\wedge$ Sent = Received $\wedge$ StoR $\in \{(m,SSN)^*\}$

$\wedge$ RtoS $\in \{(m', \neg SSN)^+ . (m,SSN)^+ + (m,SSN)^*\}$

In the above assertions, m' denotes an old message, i.e. a message for which sender has already got the ack. The StoR and RtoS mediums are described by means of regular expressions. Thus $\{(m', \neg SSN)^+ . (m,SSN)^+ + (m,SSN)^*\}$ is the set of all sequences of packets, which have either (i) a nonzero number of $(m', \neg SSN)$-packets followed by a nonzero number of (m,SSN)-packets, or, (ii) an arbitrary number of (m, SSN)-packets.

Deriving (I2)

$I \wedge R \triangleq$ Relaxed $\wedge \neg$ Sending $\equiv$ Relaxed

$I \vee R \triangleq$ Relaxed $\vee$ Sending.

$\{$Relaxed $\wedge$ RtoS $\neq$ Empty$\}$ LoseAck $\{$Relaxed$\}$

$\{$Relaxed $\wedge$ StoR $\neq$ Empty$\}$ LosePacket $\{$Relaxed$\}$

$\{$Relaxed $\wedge$ Host-Ready$\}$ SenderA_1 $\{$Sending$\}$

$\{$Relaxed $\wedge$ StoR$\neq$Empty$\}$ Receiver$_2$ $\{$Relaxed$\}$

The other transitions are not enabled and cannot occur.

$\vdash$ Relaxed $\supset \neg$ Sending $\Box$ Relaxed.

(I3), (I4), (I5) can be derived similarly by considering all transitions.

From (I2) to (I5) and Init $\supset$ Relaxed, follows

(I1) Init $\supset$ $\Box$(Relaxed $\vee$ Sending $\vee$ Send-Complete $\vee$ Acking).

These five assertions can be used to derive other required safety properties.

For example-

If there are no outstanding messages (i.e. transmitted messages for which an ack has not been received), then the sequence of messages sent is identical to the sequence of messages received. From (I1) and (sending $\vee$ SendComplete $\vee$ Acking $\supset$ Pending $\neq$ NIL), it follows that the above property can be expressed as

Relaxed $\supset$ Sent = Received .

This is indeed true.

3.2.2 The Liveness Properties:

The Liveness Properties are such as to ensure that the Alternating Bit protocol system does indeed cycle through the four states.

They are

(L1) Relaxed $\wedge$ Host Ready $\supset$ ∇ Sending.

(L2) Sending $\supset$ ∇ Send Complete.

(L3) Send-complete $\supset$ ∇ Acking.

(L4) Acking $\supset$ ∇ Relaxed.

Each liveness property depends on the occurrence of one particular transition. Thus (L1) depends on SenderA_1, (L2) depends on Receiver_1, (L3) depends on Receiver_3 and (L4) depends on SenderA_2.

Properties (L1) and (L3) are derived by a single application of the ESC rule for each property.

For (L2) and (L4), derivation is more complex.

Let an oldpacket (oldack) be defined as a packet (ack) with sequence number $\neg$SSN.

The medium StoR must be cleared of oldpackets before transition $Receiver_1$ can occur. Similarly, medium RtoS must be cleared of oldacks before transition $SenderA_2$ can occur.

The clearing of oldpackets/oldacks still does not guarantee that the required transitions must occur, because of the actions of Losepacket/LoseAck.

This motivates the following constraint on the transmission mediums. The constraint may be regarded as a restriction on the behaviour of Lose Packet/Lose Ack

Medium Constraint: $\Box\nabla$(a packet/ack is transmitted) $\supset \nabla\Box$(The medium StoR/RtoS is not Empty)

-if an unbounded number of packets/acks is transmitted, then eventually the queue of packets/acks remains forever nonempty

Note: The constraint may be expressed formally as,

$\Box\nabla$ at (Transmit) $\supset \nabla\Box$ Medium $\neq$ Empty,

where 'Transmit' is a dummy location introduced by concatenating a 'Null' transition to the 'send a packet' transition.

$\bigcirc \xrightarrow{c_\alpha \rightarrow [\text{Send}]} \bigcirc$ is changed to $\bigcirc \xrightarrow{c_\alpha \rightarrow [\text{Send}]} \bigcirc \xrightarrow{\text{True} \rightarrow [\text{Skip}]} \bigcirc$

'Transmit'
location

The constraint given is a weak constraint. It may be interpreted as - If an unbounded number of packets is transmitted, then some packet must be received. Thus, there is no requirement that the number of packets lost is bounded.

In the liveness derivations, the requirement B of the ESC rule is adapted, by dropping the antecedent term

$\chi(r_i(\bar{\pi}); f_i(\bar{y}))$. Recall that requirement B of ESC rule is,

at $l \wedge c_i(\bar{y}) \wedge \phi(\bar{\pi};\bar{y}) \wedge \chi(\bar{\pi};\bar{y}) \wedge \chi(r_i(\bar{\pi}); f_i(\bar{y})) \supset \psi(r_i(\bar{\pi}); f_i(\bar{y}))$

- any of the α_i, $i=1,\ldots,k$, transitions from location l, that preserves χ and is initiated with ϕ true, achieves ψ. This adaptation is justified, because anything implied by the weaker antecedent (i.e. the antecedent with the last term dropped) is also implied by the stronger antecedent (i.e. the original antecedent).

Derivation of L1. $\models$ (Relaxed $\wedge$ Host-Ready) $\wedge \Box \neg$Sending $\supset \Diamond$ Sending.

The ESC rule is applied to the SenderA-location.

The requirements are

A: (Relaxed $\wedge$ Host-Ready) is Sending-invariant.

For every transition α,

$\{$Relaxed $\wedge$ Host-Ready$\}$ α $\{$(Relaxed $\wedge$ Host-Ready) $\vee$ Sending$\}$

- follows from derivation of (I2).

B: For the transition SenderA$_1$ as α,

$$\text{Pending=NIL} \wedge (\text{Relaxed} \wedge \text{Host-Ready}) \wedge \neg\text{Sending} \supset [\text{Sending}(r_\alpha(\bar{y});\ f_\alpha(\bar{y}))]$$

The implication is indeed true

C: (Relaxed $\wedge$ Host-Ready) $\supset$ (Pending=NIL $\wedge$ Host-Ready) also holds.

$\models$ Relaxed $\wedge$ Host Ready $\wedge$ $\Box\neg$Sending $\supset$ ∇ Sending (by ESC)

which is equivalent to

(L1) $\models$ Relaxed $\wedge$ Host Ready $\supset$ ∇ Sending.

Derivation of (L3)

$\models$ Send Complete $\wedge\Box\neg$Acking $\supset$ ∇ Acking

The ESC rule is applied to the Receiver-Location.

The requirements are

A: Send Complete is $\neg$Acking-invariant.

Follows from (I4)

B: For the transition α = Receiver$_3$,

$$\text{Send-complete} \wedge \text{ReceiverBuffer} \neq \text{NIL} \wedge \neg\text{Acking} \supset [\text{Acking}(\bar{r}_\alpha(\bar{\pi});\ f_\alpha(\bar{y}))],$$ holds.

C: SendComplete $\supset$ ReceiverBuffer $\neq$ NIL, also holds.

Hence, by ESC rule

$\models$ send-Complete $\wedge\Box\neg$Acking $\supset$ ∇ Acking

which is equivalent to

$\models$ SendComplete $\supset$ ∇ Acking.

The argument used to derive (L2) is as follows. Eventually StoR is cleared of oldpackets. After that, a packet is transmitted into StoR an unbounded number of times by SenderB, hence (by constraint) eventually StoR remains nonempty. When this is the case, transition $receiver_1$ is enabled and must occur (by ESC rule) - thus making (L2) true.

The argument for (L4) is similar. Firstly RtoS is cleared of oldacks.

Now consider the Sender-Receiver medium, StoR. Sender-B transmits an unbounded number of messages into StoR, so that eventually StoR remains nonempty. Hence, transition $Receiver_2$ is enabled and must occur. The above reasoning holds as long as Acking is true -so that transition $Receiver_2$ occurs an unbounded number of times.

Considering medium RtoS again, (which has no oldacks), it must eventually remain nonempty (by constraint), because each transition $Receiver_2$ puts a new ack into RtoS.

If RtoS is nonempty and has no oldacks, transition $SenderA_2$ must occur - showing (L4) to be true.

Let # old-packets return the number of oldpackets in the medium StoR, that is

Oldpackets (StoR) = The number of packets in StoR with sequence number $\neg$ SSN.

Oldpackets is an abbreviation for # oldpackets (StoR).

Similarly # Oldacks (RtoS) returns the number of Oldacks in the medium RtoS.

Two additional properties are of use

(Lp) $\Box$ Sending $\supset \nabla \Box$(Sending $\wedge$ $\#$ Old Packets $=0$).

(La) $\Box$ Acking $\supset \nabla \Box$(Acking $\wedge$ $\#$ Old acks$=0$).

These properties can be derived under the assumptions

(s) Sending $\supset \exists$n: ($\#$Oldpackets = n).

(A) Acking $\supset \exists$n: ($\#$Oldacks = n).

That is, the number of oldpackets in Sending state is always finite, and the number of oldacks in Acking state is always finite.

Property (Lp) will be derived - the derivation for (La) is exactly similar.

First, using induction on the number of oldpackets, it is shown that

$$\models \text{Sending} \supset \nabla (\text{Sending} \wedge \#\text{Oldpackets} = 0).$$

Next, it can easily be derived that

$$\models \#\text{Oldpackets}= 0 \wedge \Box\, \text{Sending} \supset \Box\, (\#\, \text{Oldpackets}=0)$$

i.e. ($\#$Oldpackets=0) is Sending-invariant.

From the above two steps, (Lp) follows.

Induction:

(Basis) $\models (\text{Sending} \wedge \#\, \text{Oldpackets}=0) \supset (\text{Sending} \wedge \#\, \text{Oldpackets}=0)$

obvious.

(Induction Step)

$$\models (\text{Sending} \wedge \#\text{Oldpackets}=n+1) \supset \nabla\, (\text{Sending} \wedge \#\text{Oldpackets}=n)$$

This is derived by applying the ESC rule to the Receiver-location, and seeing that transition Receiver_2 does occur, so that

$$\models (\text{Sending} \wedge \#\text{Oldpackets}=n+1) \wedge \Box(\neg\text{Sending} \vee \#\text{Oldpackets} \neq n) \supset \nabla(\text{Sending} \wedge \#\text{Oldpackets} = n).$$

The requirements are,

A: $(\text{Sending} \wedge \#\text{Oldpackets}=n+1)$ is $(\neg\text{Sending} \vee \#\text{Oldpackets}\neq n)$ - invariant.

That is, for every transition α,

$$\{\text{Sending} \wedge \#\text{Oldpackets}=n+1\}\ \alpha\ \{\text{Sending} \wedge (\#\text{Oldpackets}=n \vee \#\text{Oldpackets}=n+1)\}$$

From the derivation of (I2), there are six transitions enabled with Sending true-LoseAck, LosePacket, SenderA_3, SenderB, Receiver$_1$ and Receiver$_2$.

LoseAck, Sender A_3 do not affect StoR medium. Receiver$_1$ is not enabled with $\#\text{Oldpackets} > 0$. For the other three transitions,

$$\{\text{Sending} \wedge \#\text{Oldpackets}=n+1\}\ \text{Losepacket}\ \{\text{Sending} \wedge \#\text{Oldpackets}=n\}.$$
$$\{\text{Sending} \wedge \#\text{Oldpackets}=n+1\}\ \text{SenderB}\ \{\text{Sending} \wedge \#\text{Oldpackets}=n+1\}.$$
$$\{\text{Sending} \wedge \#\text{Oldpackets}=n+1\}\ \text{Receiver}_2\ \{\text{Sending} \wedge \#\text{Oldpackets}=n\}.$$

Hence requirement A holds.

B: For transition α = Receiver$_2$,

$$(\text{Sending} \wedge \#\text{Oldpackets}=n+1) \wedge \text{StoR} \neq \text{Empty} \wedge \neg(\text{Sending} \wedge \#\text{Oldpackets}=n) \supset$$
$$[\text{Sending}(r_\alpha(\bar{\pi}); f_\alpha(\bar{y})) \wedge \#\text{Oldpackets}(f_\alpha(\bar{y})) = n],$$

this implication holds.

C: $(\text{Sending} \wedge \#\text{Oldpackets}=n+1) \supset \text{StoR} \neq \text{Empty}$

Hence by ESC rule,

$$\models (\text{Sending} \wedge \#\text{Oldpackets}=n+1) \wedge \Box(\neg\text{Sending} \vee \#\text{Oldpackets}\neq n) \supset \nabla(\text{Sending} \wedge \#\text{Oldpackets}=n)$$

This is equivalent to

$$\models (\text{sending} \wedge \#\text{Oldpackets}=n+1) \supset \nabla(\text{Sending} \wedge \#\text{Oldpackets}=n),$$

the required induction step.

By induction,

$$\models (\text{sending} \wedge \exists n{:}\#\text{Oldpackets}=n) \supset \nabla(\text{Sending} \wedge \#\text{Oldpackets}=0).$$

Hence, by assumption (s),

$$\models \text{Sending} \supset \nabla(\text{Sending} \wedge \#\text{Oldpackets}=0).$$

($\#$ Oldpackets=0) is sending-invariant, can be seen by a derivation similar to that of (I2). That is, for all transitions α,

$$\{\text{Sending} \wedge \#\text{Oldpackets}=0\}\ \alpha\ \{\neg\text{Sending} \vee \#\text{Oldpackets}=0\},$$

does indeed hold.

From this follows

(Lp) $\models \Box\,\text{Sending} \supset \nabla\Box(\text{Sending} \wedge \#\text{Oldpackets}=0).$

Derivation of (L2):

(1) $\models \text{Sending} \supset \Box\,\text{Sending} \vee \nabla\,\text{Send Complete}$ .... from (I3)

(2) $\models \Box\,\text{Sending} \supset \nabla\Box(\text{Sending} \wedge \#\text{Oldpackets}=0)$ ...Property (Lp)

(3) $\models \Box\,\text{Sending} \supset \Box(\text{Pending}\neq\text{NIL})$ ...from $(\text{Sending} \supset \text{Pending}=(m,\text{SSN}))$

(4) $\models \Box(\text{pending} \neq \text{NIL}) \supset \nabla(\text{Sender-B occurs})$ ....by ESC rule applied to Sender B location.

(5) $\models \Box\,\text{Sending} \supset \nabla\,(\text{Sender-B occurs})$ .... by (3), (4)

(6) $\models \Box\,\text{Sending} \supset \Box\nabla(\text{Sender-B occurs})$ .... by (5), Temporal Logic reasoning

(7) $\models \Box\nabla(\text{Sender B occurs}) \supset \nabla\Box(\text{StoR} \neq \text{Empty})$ .... by Constraint on StoR

(8) $\models \Box\,\text{Sending} \supset \nabla\Box(\text{StoR} \neq \text{Empty})$ .....by (6), (7)

(9) $\models \Box\,\text{Sending} \supset \nabla\Box(\text{StoR} \neq \text{Empty}) \wedge \nabla\Box(\text{Sending} \wedge \#\text{old-packets}=0)$ ...by (2),(8)

(10) $\models \Box\,\text{Sending} \supset \nabla\Box(\text{Sending} \wedge \text{StoR} \neq \text{Empty} \wedge \#\text{old-packets}=0)$

.....by (9), Temporal Logic Theorem

(11) $\models (\text{Sending} \wedge \text{StoR} \neq \text{Empty} \wedge \#\text{oldpackets}=0) \supset \text{StoR} \in \{(m,\text{SSN})^{+}\}$

.....by Sending

(12) $\models \Box\,\text{Sending} \supset \nabla\Box(\text{StoR} \in \{(m,\text{SSN})^{+}\})$ .....by (10),(11)

(13) $\models \Box\,\text{Sending} \supset \nabla[\Box(\text{StoR} \in \{(m,\text{SSN})^{+}\}) \wedge \text{Sending}]$

....by (12), Temporal Logic Theorem

(14) $\models \text{Sending} \wedge \Box(\text{StoR} \in \{(m,\text{SSN})^{+}\}) \supset \nabla\,\text{SendComplete}$

....by ESC rule applied to Receiver-Location

(15) $\models \Box\,\text{Sending} \supset \nabla\,\text{SendComplete}$ ....by (13), (14)

(L2) (16) $\models \text{Sending} \supset \nabla\,\text{SendComplete}$ ....by (1), (15)

Proceedings in a similar manner, (L4) may also be derived.

(L4) $\models \text{Acking} \supset \nabla\,\text{Relaxed}$.

CONCLUSION

Comparing the methods for deriving safety and liveness properties, we feel that [LAM3] is the best method for safety properties, and [OL] is the best method for liveness properties.

The following reasons can be given for the superiority of [LAM3] over [MP] and [OG].

(i) It is an axiomatic method, and not based on any operational model.

(ii) It can be used to examine programs with nested cobegins- unlike [MP], wherein programs have a fixed number of processes.

(iii) The indivisible actions can be at any level - unlike [OG], [MP], wherein indivisible actions are fixed at the assignment statement/expression level.

(iv) Explicit use of location predicates - in [OG] Auxiliary Variables must be introduced.

(v) The semantics of processes is given by process-invariants and not input-output behaviour as in [OG]. The [LAM3] formula $\{P\}\ S_1\ \{P\}$, for process S_1 within a cobegin statement cobegin $S_1 \| S_2 \| \ldots \| S_n$ coend, expresses a process-invariant P maintained throughout S_1. The corresponding [OG] formula $\{P_1\}\ S_1\ \{Q_1\}$ is an input-output assertion. The disadvantage of [OG] is that, if S_1 is replaced by an 'equivalent' process S_1', in addition to deriving $\{P_1\}\ S_1'\ \{Q_1\}$, it must be shown that $\{P_1\}\ S_1'\ \{Q_1\}$ is interference-free from each of the other proof-outlines $\{P_i\}\ S_i\ \{Q_i\}$, $2 \leqslant i \leqslant n$. For [LAM3] only $\{P\}\ S_1'\ \{P\}$ must be derived.

Among the methods for deriving liveness properties, [MP], [OL] and [LAM1] we feel that [OL] is superior to the other methods because

(i) It is an axiomatic method. The other two methods use operational models.

(ii) It can examine programs with nested cobegins-unlike the other two methods.

(iii) It is temporal logic based-hence the formalism and theorems of temporal logic are ready to hand.

(iv) Proofs are derived using proof-lattices-this facilitates high-level, informal reasoning without sacrificing rigour.

We feel that a fully formal step-by-step proof, of even small programs, is incomprehensible. The effort spent in developing such a proof is not worth the resulting gain in understanding of the program. Informal reasoning and semi-formal methods must play a large part, if program proofs are to remain manageable.

Most properties of interest of concurrent programs follow from a few safety invariants and 'bounding functions' for termination. These safety invariants and bounding functions must be discovered by examining the program. Subsequently, they can be justified informally or derived by semi-formal checking of relevant indivisible actions. After this, the derivation of further properties of interest, becomes a simple matter.

The above approach is analogous to deriving the loop-invariants only for a sequential program-not the step-by-step Hoare-logic proof.

Some suggested directions for further work are

(i) Extension of Temporal Logic based methods to derive safety properties not expressible by monadic $\Box$, (eg. First Come First Served has the form $P \supset R \Box Q$).

(ii) Extension of formal methods of treating concurrent programs to handle local variables of processes. That is, specify the semantics of declarations within a block.

REFERENCES

List of Abbreviations used

CACM Communications of the ACM

JACM Journal of the ACM

LNCS Lecture Notes in Computer Science,Springer-Verlag

TOPLAS ACM Transactions on Programming Languages and Systems

[BMP] Ben-Ari M., Manna Z., Pnueli A.,
The Temporal Logic of branching time
Proceedings of the Eighth ACM Symposium on Principles
of Programming Languages, p.169-176, Jan.1981.

[BOC] Bochmann G.V.,
Hardware Specification with Temporal Logic: An Example
IEEE Transactions on Computers, C-31(3), p.223-232,March 1982.

[DLP] De Millo R.A., Lipton R.J., Perlis A.J.,
Social Processes and Proofs of Theorems and Programs
CACM 22(5), p.271-280, May 1979.

[DIJ 1] Dijkstra E.W.,
Heirarchical Ordering of Sequential Processes
Acta Informatica 1(2), p.115-138, 1971.
Operating Systems Techniques, edited by Hoare, C.A.R.,
Perrott R.H.,
p.72-93, Academic Press.

[DIJ 2] Dijkstra E.W., Lamport L., Martin A.J.,Scholten C.S.,
Steffens E.F.M.,
On-the-fly Garbage Collection: An exercise in cooperation
CACM 21(11), p.966-975, November 1978.

[DIJ 3] Dijkstra E.W.,
A personal summary of the Gries-Owicki Theory, EWD 554
Selected Writings on Computing: A Personal Perspective,
P.188-199, Springer-Verlag.

[DIJ 4] Dijkstra E.W.,
A Discipline of Programming
Prentice Hall-India.

[GPSS] Gabbay D., Pnueli A., Shelah S., Stavi J.,
On the temporal analysis of fairness
Proceedings Seventh ACM Symposium on Principles of Programming Languages, P.167-173, 1980.

[GRI 1] Gries, D.,
An exercise in proving parallel programs correct
CACM 20(12), P.921-930, December 1977.

[GRI 2] Gries D.,
Is Sometime ever better than Alway?
TOPLAS 1(2), P.258-265, October 1979.

[GUP] Gupta A.,
Temporal Logic of Programs
B.Tech. Project Report, B.Tech.-CS-83-11, IITK.,April 1983.

[HO] Hailpern B.T., Owicki S.,
Verifying Network Protocols using Temporal Logic
Proceedings Trends and Applications 1980: Computer Network Protocols (Gaithersburg), IEEE Computer Society, P. 18-28, May 1980.

[HOA] Hoare C.A.R.,
An axiomatic basis for computer programming
CACM 12(10), p.576-580, October 1969.

[HGLS] Holt R.C., Graham G.S., Lazowska E.D., Scott M.A.,
Structured Concurrent Programming with Operating Systems Applications
Addison-Wesley.

[KEL] Keller R.M.,
Formal Verification of Parallel Programs
CACM 19(7), p.371-384, July 1976.

[LAM1] Lamport L.,
Proving the correctness of multiprocess programs
IEEE Transactions on Software Engineering SE-3(2),
P.125-143, March 1977.

[LAM2] Lamport L.,
"Sometime" is sometimes "Not Never": On the Temporal Logic of Programs
Proceedings of the Seventh ACM Symposium on Principles of Programming Languages, P.174-185, 1980.

[LAM3] Lamport L.,
The "Hoare Logic" of Concurrent Programs
Technical Report CSL-79, SRI International, October 1978.

[LAM4] Lamport L.,
A new approach to proving the correctness of Multiprocess Programs
TOPLAS 1(1), p.84-97, July 1979.

[LIP] Lipton R.J.,
Reduction: A method of proving properties of Parallel Programs
CACM 18(12), p.717-721, December 1975.

[MW] Manna Z., Waldinger K.,
Is "Sometime" sometimes better than "Always"?
CACM 21 (2), p.159-172, February 1978.

[MP] Manna Z., Pnueli A.,
Verification of Concurrent Programs: Temporal Proof Principles, Technical Report CS 81-23, Department of Applied Mathematics, Weizmann Institute of Science, Sept. 1981.

[OG] Owicki S., Gries D.,
An axiomatic proof technique for Parallel Programs
Acta Informatica 6(4), p.319-340, 1976
Programming Methodology: A Collection of articles by members of IFIP WG2.3, edited by Gries D., p.130-152, Springer-Verlag.

[OG2] Owicki S., Gries D.,
Verifying Properties of Parallel Programs: An axiomatic approach
CACM 19(5), p.279-285, May 1976.

[OL] Owicki S., Lamport L.,
Proving Liveness Properties of Concurrent Programs
TOPLAS 4(3), p.455-495, July 1982.

[PET] Peterson G.L.,
Myths About Mutual Exclusion
Information Processing Letters 12(3),
p.115-116, 13 June 1981.

[PNU] Pnueli A.,
The temporal semantics of concurrent programs
LNCS 70: Semantics of Concurrent Computations, p.1-20, 1979

[SUN] Sunshine C.A., Thompson D.H., Erickson R.W., Gerhart S.L, Schwabe D.,
Specification and Verification of Communication Protocols in AFFIRM using State Transition models
IEEE Transactions on Software Engineering, SE 8(5),
p.460-489, Sep. 1982.